(12) NACH DEM VERTRAG ÜBER DIE INTERNATIONALE ZUSAMMENARBEIT AUF DEM GEBIET DES PATENTWESENS (PCT) VERÖFFENTLICHTE INTERNATIONALE ANMELDUNG

(19) Weltorganisation für geistiges Eigentum
Internationales Büro

(43) Internationales Veröffentlichungsdatum
27. Dezember 2002 (27.12.2002)

PCT

(10) Internationale Veröffentlichungsnummer
WO 02/103532 A2

(51) Internationale Patentklassifikation[7]: G06F 13/28, 13/36, G11C 16/00, G06F 15/80, 9/45, 11/36, 15/18, 1/04, 9/38, 15/76, 11/22, 15/177, 7/00, 1/32

(21) Internationales Aktenzeichen: PCT/EP02/06865

(22) Internationales Anmeldedatum:
20. Juni 2002 (20.06.2002)

(25) Einreichungssprache: Deutsch

(26) Veröffentlichungssprache: Deutsch

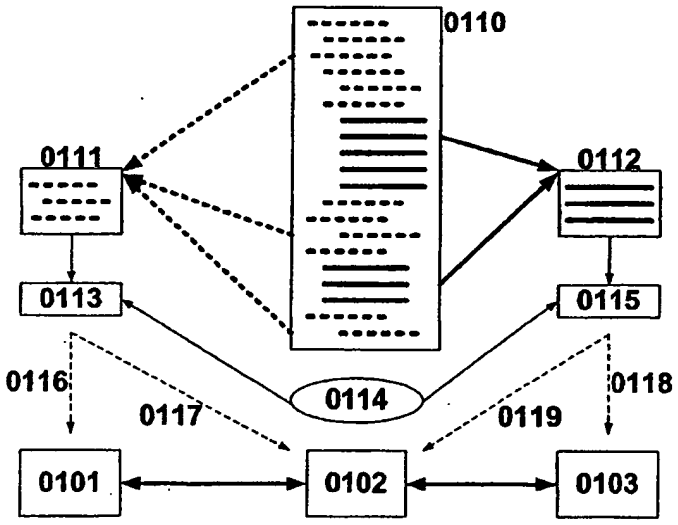
(30) Angaben zur Priorität:

101 29 237.6	20. Juni 2001 (20.06.2001)	DE
01115021.6	20. Juni 2001 (20.06.2001)	EP
101 35 210.7	24. Juli 2001 (24.07.2001)	DE
101 35 211.5	24. Juli 2001 (24.07.2001)	DE
101 39 170.6	16. August 2001 (16.08.2001)	DE
101 42 231.8	29. August 2001 (29.08.2001)	DE
101 42 903.7	3. September 2001 (03.09.2001)	DE
101 42 894.4	3. September 2001 (03.09.2001)	DE
101 42 904.5	3. September 2001 (03.09.2001)	DE
101 44 732.9	11. September 2001 (11.09.2001)	DE
101 44 733.7	11. September 2001 (11.09.2001)	DE
101 45 792.8	17. September 2001 (17.09.2001)	DE
101 45 795.2	17. September 2001 (17.09.2001)	DE
101 46 132.1	19. September 2001 (19.09.2001)	DE
101 54 260.7	5. November 2001 (05.11.2001)	DE
101 54 259.3	5. November 2001 (05.11.2001)	DE
01129923.7	14. Dezember 2001 (14.12.2001)	EP
02001331.4	18. Januar 2002 (18.01.2002)	EP
102 02 044.2	19. Januar 2002 (19.01.2002)	DE
102 02 175.9	20. Januar 2002 (20.01.2002)	DE
102 06 653.1	15. Februar 2002 (15.02.2002)	DE
102 06 857.7	18. Februar 2002 (18.02.2002)	DE
102 06 856.9	18. Februar 2002 (18.02.2002)	DE

[Fortsetzung auf der nächsten Seite]

(54) Title: DATA PROCESSING METHOD

(54) Bezeichnung: VERFAHREN ZUR BEARBEITUNG VON DATEN



(57) Abstract: The invention relates to a method for translating programmes into a system consisting of at least one first processor and a re-configurable unit. According to the method, parts of the code that are suitable for the re-configurable unit are determined and extracted and the remaining code is extracted in this manner for processing by the first processor.

[Fortsetzung auf der nächsten Seite]

WO 02/103532 A2

102 07 225.6	21. Februar 2002 (21.02.2002)	DE
102 07 224.8	21. Februar 2002 (21.02.2002)	DE
102 07 226.4	21. Februar 2002 (21.02.2002)	DE
102 08 435.1	27. Februar 2002 (27.02.2002)	DE
102 08 434.3	27. Februar 2002 (27.02.2002)	DE
102 12 622.4	21. März 2002 (21.03.2002)	DE
102 12 621.6	21. März 2002 (21.03.2002)	DE
02 009 868.7	2. Mai 2002 (02.05.2002)	EP

(71) **Anmelder** *(für alle Bestimmungsstaaten mit Ausnahme von US)*: **PACT XPP TECHNOLOGIES AG** [DE/DE]; Muthmannstrasse 1, 80939 München (DE).

(72) **Erfinder; und**

(75) **Erfinder/Anmelder** *(nur für US)*: **VORBACH, Martin** [DE/DE]; Gotthardstrasse 117A, 80689 München (DE). **NÜCKEL, Armin** [DE/DE]; Drosselweg 4, 76777 Neupotz (DE). **MAY, Frank** [DE/DE]; An der Tuchbleiche 12, 81927 München (DE). **WEINHARDT, Markus** [DE/DE]; Westendstrasse 154, 80339 München (DE). **CARDOSO, Joao, Manuel, Paiva** [PT/PT]; Rua Sao Joao, 10, Vila Cova a Coelheira, P-3650 Vila Nova de Paiva (PT).

(74) **Anwalt: PIETRUK, Claus, Peter**; Heinrich-Lilienfein-Weg 5, 76229 Karlsruhe (DE).

(81) **Bestimmungsstaaten** *(national)*: AE, AG, AL, AM, AT (Gebrauchsmuster), AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE (Gebrauchsmuster), DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZM, ZW.

(84) **Bestimmungsstaaten** *(regional)*: ARIPO-Patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), eurasisches Patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), europäisches Patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI-Patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Erklärung gemäß Regel 4.17:

— *hinsichtlich der Berechtigung des Anmelders, ein Patent zu beantragen und zu erhalten (Regel 4.17 Ziffer ii) für die folgenden Bestimmungsstaaten AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZM, ZW, ARIPO-Patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), eurasisches Patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), europäisches Patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI-Patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG)*

Veröffentlicht:

— *ohne internationalen Recherchenbericht und erneut zu veröffentlichen nach Erhalt des Berichts*

Zur Erklärung der Zweibuchstaben-Codes und der anderen Abkürzungen wird auf die Erklärungen ("Guidance Notes on Codes and Abbreviations") am Anfang jeder regulären Ausgabe der PCT-Gazette verwiesen.

(57) Zusammenfassung: Die Erfindung betrifft ein Verfahren zur Übersetzung von Programmen auf ein System bestehend aus wenigstens einem ersten Prozessor und einer rekonfigurierbaren Einheit. Hierbei ist vorgesehen, dass die Codeteile, die für die rekonfigurierbare Einheit geeignet sind, bestimmt und extrahiert werden und der verbleibende Code zur Abarbeitung durch den ersten Prozessor derart extrahiert wird.

Titel: Verfahren zur Bearbeitung von Daten

Beschreibung

Die vorliegende Erfindung befaßt sich mit Datenverarbeitung. Insbesondere befaßt sich die vorliegende Erfindung mit herkömmlichen, d.h. konventionellen und rekonfigurierbaren Prozessorarchitekturen sowie mit Verfahren hierfür, die eine Übersetzung einer klassischen Hochsprache (PROGRAMM) wie Pascal, C, C++, Java, etc. ermöglichen, insbesondere auf eine rekonfigurierbare Architektur. Insbesondere befaßt sich die vorliegende Erfindung mit der Integration und/oder engen Kopplung von rekonfigurierbaren Prozessoren mit Standardprozessoren, dem Datenaustausch und der Synchronisation der Datenverarbeitung.

Unter einer konventionellen Prozessorarchitektur (PROZESSOR) werden vorliegend beispielsweise sequentielle Prozessoren mit einer von-Neumann- oder Harvardarchitektur verstanden, wie z.B. Kontroller, CISC-, RISC-, VLIW-, DSP-, u.ä. Prozessoren verstanden.

Unter einer rekonfigurierbaren Zielarchitektur werden vorliegend Bausteine (VPU) mit wiederholt und insbesondere zur Laufzeit insbesondere unterbrechungsfrei konfigurierbarer Funktion und/oder Vernetzung verstanden, insbesondere integrierte Bausteine mit einer Mehrzahl von ein- oder mehrdimensional angeordneten arithmetischen und/oder logischen und/oder analogen und/oder speichernden insbesondere evtl. auch grobgranularen Baugruppen (PAE), die direkt oder durch ein Bussystem miteinander verbunden sind.

Zur Gattung dieser Bausteine zählen insbesondere systolische Arrays, neuronale Netze, Mehrprozessor Systeme, Prozessoren mit mehreren Rechenwerken und/oder logischen Zellen, Vernetzungs- und Netzwerkbausteine wie z.B. Crossbar-Schalter, ebenso wie bekannte Bausteine der Gattung FPGA, DPGA, XPUTER, etc.. Hingewiesen wird insbesondere in diesem Zusammenhang auf die folgenden Schutzrechte desselben Anmelders: P 44 16 881.0-53, DE 197 81 412.3, DE 197 81 483.2, DE 196 54 846.2-53, DE 196 54 593.5-53, DE 197 04 044.6-53, DE 198 80 129.7, DE 198 61 088.2-53, DE 199 80 312.9, PCT/DE 00/01869, DE 100 36 627.9-33, DE 100 28 397.7, DE 101 10 530.4, DE 101 11 014.6, PCT/EP 00/10516, EP 01 102 674.7, DE 196 51 075.9-53, DE 196 54 846.2-53, DE 196 54 593.5-53, DE 197 04 728.9, DE 197 07 872.2, DE 101 39 170.6, DE 199 26 538.0, DE 101 42 904.5, DE 101 10 530.4. Diese sind hiermit zu Offenbarungszwecken vollumfänglich eingegliedert.

Das System kann insbesondere als (Standard)-Prozessor oder Baugruppe ausgestaltet sein und/oder in einem Halbleiter (System on Chip SoC) integriert sein.
Rekonfigurierbare Bausteine (VPUs) unterschiedlicher Gattungen (wie z.B. PACT XPP-Technologie, Morphics, Morphosys, Chamele-

on) sind zu bestehenden technischen Umgebungen und Programmierverfahren weitgehend inkompatibel.

Programme für diese Bausteine sind typisch inkompatibel zu bereits bestehenden Programmen von CPUs. Dadurch wird ein erheblicher Entwicklungsaufwand zur Programmierung erforderlich, z.B. besonders für Bausteine der Gattungen Morphics, Morphosys. Chameleon integriert bereits einen Standardprozessor (ARC) auf mehr oder minder rekonfigurierbaren Bausteinen. Dadurch stehen Ansätze für Tools zur Programmierung zur Verfügung. Allerdings ist nicht jede technische Umgebung für den Einsatz von ARC-Prozessoren geeignet, insbesondere liegen bestehende Programme, Codebibliotheken etc. oftmals für beliebige unbestimmte andere CPUs vor.

Es hat sich in internen Versuchen gezeigt, daß es bestimmte Verfahren und Programmabläufe gibt, die sich besser mit einer rekonfigurierbare Architektur abarbeiten lassen als mit einer konventionellen Prozessorarchitektur. Umgekehrt gibt es auch solche Verfahren und Programmabläufe, die besser mit einer konventionellen Prozessorarchitektur ausgeführt werden können. Es ist dafür wünschenswert, um eine jeweilige Optimierung zu ermöglichen, eine Ablaufteilung vorzusehen.

Bekannte Übersetzungsverfahren für rekonfigurierbare Architekturen unterstützen keine Weitergabe von Codes an beliebige Standard-Compiler zur Generierung von Objektcodes für einen beliebigen PROZESSOR. Gewöhnlicherweise ist der PROZESSOR fest innerhalb des Compilers definiert.

Weiterhin existieren keine Scheduling-Mechanismen zur Rekonfiguration der einzelnen generierten Konfigurationen für VPUs. Insbesondere fehlen Scheduling-Mechanismen für die Konfigura-

tion unabhängiger extrahierter Teile gleichwohl wie für einzelne Partitionen extrahierter Teile. Entsprechende Übersetzungsverfahren nach dem Stand der Technik sind beispielsweise definiert durch die Dissertation „Übersetzungsmethoden für strukturprogrammierbare Rechner, Dr. Markus Weinhardt, 1997".

Zur Partitionierung von Array-CODE sind mehrere Verfahren nach dem Stand der Technik bekannt, z. B. Joao M. P. Cardoso, „Compilation of Java™ Algorithms onto Reconfigurable Computing Systems with Exploitation of Operation-Level Parallelism", Ph. D. Thesis Universidade Técnica de Lisboa (UTL), 2000.

Diese Verfahren sind jedoch in keine kompletten Compilersysteme eingebettet. Weiterhin setzen die Verfahren die vollständige Steuerung der Rekonfiguration durch einen Hostprozessor voraus, was einen erheblichen Aufwand bedeutet. Die Partitionierungsstrategien sind für FPGA-basierende Systeme ausgelegt und entsprechen daher keinem echten Prozessormodell.

Die Aufgabe dieser Erfindung besteht darin, Neues für die gewerbliche Anwendung bereitzustellen.

Die Lösung dieser Aufgabe wird in unabhängiger Form beansprucht. Bevorzugte Ausführungen finden sich in den Unteransprüchen.

Ein rekonfigurierbarer Prozessor (VPU) wird somit in eine technische Umgebung eindesigned, die einen Standardprozessor (CPU) besitzt, wie beispielsweise einen DSP, RISC, CISC-Prozessor oder (Mikro)-Kontroller aufweist. Das Design kann erfindungsgemäß derart erfolgen, dass eine einfache und leistungsfähige Anbindung besteht. Ein sich ergebender Aspekt ist die einfache Programmierbarkeit des entstehenden Systems. Die

Weiterverwendung bestehender Programme der CPU sowie die Codekompatibilität und die einfache Integration der VPU in die bestehenden Programme finden Berücksichtigung.

Eine VPU (oder ohne jeweils besonders erwähnt zu werden, mehrere VPUs) wird derart mit einer bevorzugten CPU (oder ohne jeweils besonders erwähnt zu werden, mehreren CPUs) gekoppelt, dass sie die Stelle und Funktion eines Coprozessors (bzw. mehrerer wahlweise ansprechbarer Coprozessoren) einnimmt. Die Funktion ermöglicht die einfache Einbindung in bestehende Programmcodes entsprechend den bereits existierenden Methoden zum Umgang mit Coprozessoren nach dem Stand der Technik.

Der erfindungsgemäße Datenaustausch zwischen CPU und VPU kann mittels Speicherkopplung und/oder IO-Kopplung erfolgen. CPU und VPU können sämtliche Ressourcen teilen, in besonderen Ausgestaltungen ist es auch möglich, dass CPU und VPU nur einen Teil der Ressourcen gemeinsam verwenden und andere Ressourcen jeweils explizit und/oder exclusive für eine CPU oder VPU zur Verfügung stehen.

Um einen Datenaustausch durchzuführen, können Datensätze und/oder Konfigurationen in jeweils besonders dafür vorgesehen Speicherbereiche kopiert bzw. geschrieben/gelesen werden und/oder entsprechende Basisadressen gesetzt werden, dass diese auf die jeweiligen Datenbereiche zeigen.

Zur Steuerung des Coprozessors wird bevorzugt ein Datensatz vorgesehen, der beispielsweise die Grundeinstellungen einer VPU beeinhaltet, wie beispielsweise bestimmte Basisadressen. Desweiteren können Statusvariablen zur Ansteuerung und Funktionssteuerung einer VPU durch eine CPU sowie für Rückmeldungen einer VPU an eine CPU vorgesehen sein. Der Datensatz kann über

einen gemeinsamen Speicher (RAM) und/oder über einen gemeinsamen peripheren Adressraum (IO) ausgetauscht werden.

Zur Synchronisation der CPU und VPU können einseitig oder gegenseitig wirkende Interruptverfahren (die z.B. durch Signaltransfer über insbesondere dedizierte bzw. hierfür ausgebildete Interruptleitungen und/oder Interrupteingänge realisiert sind) vorgesehen sein und/oder die Synchronisation erfolgt mittels Pollingverfahren. Weiterhin können Interrupts zur Synchonisation von Daten- und/oder DMA-Transfers verwendet werden.

In einer besonders zu bevorzugenden Ausgestaltung wird eine VPU durch eine CPU gestartet und arbeitet danach bevorzugt unabhängig die Applikation ab.

Besonders leistungsfähig ist ein bevorzugter Aufbau, bei welchen die verwendete VPU eigene Mechanismen zum Laden und Kontrollieren von Konfigurationen vorsieht. Zur Gattung dieser VPUs gehören beispielsweise PACT XPP und Chameleon. Die erfindungsgemäßen Schaltungen ermöglichen ein Verfahren zum Betrieb derart, dass die Konfigurationen der VPU zusammen mit dem auszuführenden Programm der CPU in einen Speicher geladen werden. Die CPU kann während der Ausführung des Programmes die VPU auf die Speicherstellen verweisen (z.B. durch Angabe der Adressen oder Pointer), die die jeweils auszuführenden Konfigurationen beinhalten. Die VPU kann daraufhin die Konfigurationen selbständig und ohne weitere Einflußnahme durch die CPU laden. Die Ausführung startet sofort oder ggf. durch eine zusätzliche Information (z.B. Interrupt und/oder Start-Befehl) durch die CPU.

In einer besonders bevorzugten Erweiterung kann die VPU selbständig innerhalb eines Speichers Daten lesen und schreiben.

In einer besonders bevorzugten Erweiterung kann die VPU ebenfalls selbständig neue Konfigurationen aus dem Speicher laden und sich bei Bedarf neu konfigurieren, ohne dass es eines weiteren Einflusses durch die CPU bedarf..

Diese Ausgestaltungen ermöglichen einen weitestgehend von CPUs unabhängigen Betrieb von VPUs. Lediglich ein Synchronisationsaustausch zwischen CPU und VPU, der bevorzugt bidirektional stattfinden kann, sollte zusätzlich vorgesehen werden, um die Datenverarbeitungen und/oder Konfigurationsausführungen aufeinander abzustimmen.

Es wurde weiter erkannt, daß Verfahren zur Datenverarbeitung bevorzugt so ausgelegt werden können und/oder sollen, daß jeweils für die rekonfigurierbare Zielarchitektur (VPU) besonders geeignete Teile (VPU-CODE) des zu übersetzenden Programmes identifiziert und extrahiert werden, um eine besonders effiziente Datenverarbeitung zu ermöglichen. Diese Teile sind entsprechend zu partitionieren und die Konfiguration der einzelnen Partitionen ist in ihrer zeitlichen Reihenfolge zu steuern.

Die verbleibenden Teile des Programmes können auf eine konventionelle Prozessorarchitektur (PROZESSOR) übersetzt werden. Dies geschieht bevorzugt dergestalt, daß diese Teile als Hochsprachencode in einer Standard-Hochsprache (z. B. ANSI C) derart ausgegeben werden, daß ein gewöhnlicher (ggf. bereits existierender) Hochsprachencompiler diese ohne weiteres verarbeiten kann.

Weiterhin sei angemerkt, daß die Verfahren auch auf Gruppen von mehreren Bausteinen angewendet werden können.

Insbesondere kann eine Art "Double-Buffering" zur besonders einfachen und zugleich schnellen Rekonfiguration angewendet werden, in welchem eine Mehrzahl von VPUs vorgesehen sind, wobei ein Teil der VPUs zu einer Zeit rekonfiguriert werden kann, zu welcher ein anderer Teil rechnet und möglicherweise ein Weiterer etwa inaktiv sein kann. Die Daten-, Trigger-, Statusverbindungen etc. werden zwischen der Mehrzahl von VPUs geeignet ausgetauscht und ggf. durch adressierte Busse und/oder Multiplexer/Demultiplexer entsprechend der aktuell aktiven und/oder zu rekonfigurierenden VPUs verschaltet.

Ein Vorteil dieses Verfahrens liegt darin, daß bestehender Code, der für einen beliebigen PROZESSOR geschrieben wurde, unter Einbeziehung einer VPU weiterverwendet werden kann und keine oder nur vergleichsweise geringe Modifikationen durchgeführt werden müssen. Die Modifikationen können zudem schrittweise erfolgen, wobei nach und nach immer mehr Code von dem PROZESSOR auf die VPU übertragen werden kann. Das Projektrisiko sinkt und die Überschaubarkeit steigt wesentlich an. Es wird darauf hingewiesen, daß einer derartigen sukzessive Übertragung von immer mehr Aufgaben auf die VPU, d.h. auf das integrale multidimensionale partiell rekonfigurierbaren insbesondere grobgranulare Feld an Elementen, eine besondere Bedeutung für sich hat und für sich als erfinderisch angesehen wird aufgrund seiner gravierenden Vorteile bei der Systemportierung.

Weiterhin kann der Programmierer in seiner gewohnten Entwicklungsumgebung arbeiten und muß sich nicht auf eine neue, möglicherweise fremde Entwicklungsumgebung einstellen.

Ein erster wesentlicher Aspekt der vorliegenden Erfindung ist darin zu sehen, daß ein PROZESSOR derart mit einer oder mehreren VPU(s) verbunden wird, daß ein effizienter Informationsaustausch, insbesondere in Form von Daten- und Statusinformation möglich ist.

Der Anordnung eines herkömmlichen Prozessors und eines rekonfigurierbaren Prozessors, dergestalt, daß ein Austausch von Daten- und/oder Statusinformation zwischen denselben während der Abarbeitung eines oder mehrerer Programme möglich ist und/oder ohne daß insbesondere die Datenverarbeitung auf dem rekonfigurierbaren Prozessor und/oder dem herkömmlichen Prozessor signifikant unterbrochen werden muß, sowie der Ausbildung eines derartigen Systems, wird gleichfalls für sich Bedeutung zugemessen.

Es können zunächst beispielsweise eines oder alle der folgenden Verbindungsverfahren und/oder -mittel verwendet werden:

a) Shared-Memory
b) Netzwerk (beispielsweise Bussysteme wie z.B. PCI-Bus, Serielle Busse wie z.B. Ethernet)
c) Kopplung an einen internen Registersatz oder mehrere interen Registersätze
d) andere Speichermedien (Festplatte, Flash-ROM, etc.)

Prinzipiell kann auch die VPU und/oder CPU selbständig ohne Zuhilfenahme eines DMAs auf den Speicher zugreifen. Der gemeinsame Speicher kann insbesondere auch als Dualport- oder Multiportspeicher ausgestaltet sein. Dem System können weitere Baugruppen zugeordnet werden, insbesondere können rekonfigurierbare FPGAs eingesetzt werden, um eine feingranulare Verarbeitung von einzelner Signale oder Datenbits zu ermöglichen

und/oder flexible adaptierbare Interface (z.B. diverse serielle Schnittstellen (V24, USB, etc.), diverse parallele Schnittstellen, Festplattenschnittstellen, Ethernet, Telekommunikationsschnittstellen (a/b, T0, ISDN, DSL, etc)) aufbauen zu können.

Der Aufbau einer VPU ist beispielsweise bekannt aus den o. g. zitierten Anmeldungen. Versuche zu alternativen Bausteindefinitionen sind beispielsweise unter dem Namen Chameleon geführt worden. VPUs lassen sich auf unterschiedliche Weise in ein System integrieren. Ein Anschluß an einen Hostprozessor ist beispielsweise möglich. Je nach Verfahren kann der Hostprozessor die Konfigurationskontrolle (HOSTRECONF) mit übernehmen (z. B. Chameleon) oder, z. B., eine dedizierte Einheit (CT) zur Steuerung der (Re)Konfiguration bestehen.

Entsprechend generiert der Übersetzer gemäß dem beschriebenen Verfahren die Steuerinformation für die Rekonfiguration für eine CT und/oder einen HOSTRECONF.

Es kann nun das Übersetzungsprinzip derart ausgestaltet sein, daß aus einem PROGRAMM mittels eines PRÄPROZESSORS die Teile extrahiert werden, die sich auf die jeweils bestimmte(n) VPU(s) effizient und/oder sinnvoll abbilden lassen. Diese Teile werden in ein für VPUs geeignetes Format transformiert (NML) und dann weiter in einen Objektcode übersetzt.

Der verbleibenden Code und/oder der extrahierte Code wird erfahrungsgemäß an oder bezüglich der Stelle der durch die Extraktion fehlenden Code-Teile um einen Interface-Code erweitert, der entsprechend der Architektur des Zielsystems die Kommunikation zwischen PROZESSOR(en) und VPU(s) steuert. Der verbleibende und ggf. erweiterte Code kann bevorzugt

trahiert werden sollen. Beispielsweise kann dies folgendermaßen erfolgen:

```
...
Code
...
# START_EXTRACTION
Zu extrahierender Code
# END_EXTRACTION
...
Code
...
```

„// START_EXTRACTION" kennzeichnet den Beginn eines zu extrahierenden Codes.
„// END_EXTRACTION" kennzeichnet das Ende eines zu extrahierenden Code.

In einem solchen Fall ist die Einheit zur Umsetzung des Programms in Konfigurationscodes dazu ausgebildet, die Hints beziehungsweise Umsetzungsvorgaben zu erkennen.

Es ist auch möglich, daß zur Extraktion durch Aufruf von NML-Routinen Teile des PROGRAMMES direkt in NML implementiert werden und in die NML-Routinen durch Aufrufe (calls) gesprungen wird. Beispielsweise erfolgt dies derart:

a) NML-Code

```
...
procedure EXAMPLE
begin
...
end
...
```

b) PROGRAMM Code

```
--
Code
--
call EXAMPLE          // Aufruf des NML-Codes
--
Code
--
```

In diesem Fall ist die Einheit zur Umsetzung dazu ausgebildet, NML-Programmteile, das heißt Programmteile zur Ausführung in und/oder auf einem rekonfigurierbaren Array in ein größeres Programm einzubinden.

Es ist weiter alternativ und/oder zusätzlich eine Extraktion aus einer objektorientierten Klasse möglich. Für eine VPU geeignete Makros werden als Klasse in der Klassenhierarchie einer objektorientierten Programmiersprache definiert. Die Makros können dabei durch Annotation derart gekennzeichnet sein, daß sie als für eine VPU bestimmte Codes erkannt und entsprechend - auch in höheren Hierarchien der Sprache - weiterverarbeitet werden.

Innerhalb eines Makros ist bevorzugt eine bestimmte Vernetzung und/oder Abbildung durch das Makro vorgegeben, die sodann die Abbildung des Makros auf die VPU bestimmt.

Durch die Instantiierung und Verkettung der Klasse entsteht eine Implementierung der Funktion, bestehend aus mehreren Makros auf der VPU. Mit anderen Worten definiert die Instantiierung und Verkettung der Makros die Abbildung und Vernetzung der einzelnen Operationen aller Makros auf der VPU und/oder

ggf. die Vernetzung und/oder den Datenaustausch zwischen VPU und CPU.

Die Interfacecodes werden bei der Instantiierung hinzugefügt. Die Verkettung beschreibt das detaillierte Mapping der Klasse auf die VPU.

Eine Klasse kann beispielsweise auch als ein Aufruf einer oder mehrerer NML-Routinen gebildet werden.

a) Klassen-Code

```
...
class EXAMPLE
begin
...
end
...
```

b) PROGRAMM Code

```
...
Code
...
EXAMPLE var()              // Instantiierung der Klasse
...
Code
...
```

Es ist weiter auch eine Extraktion durch Analyse möglich. Durch an die jeweilige VPU angepaßte Analysemethoden werden Teile innerhalb des PROGRAMMES erkannt, die effizient und/oder sinnvoll auf die VPU abbildbar sind.
Diese Teile werden aus dem PROGRAMM extrahiert.

Eine beispielsweise für viele VPUs geeignete Analysemethode ist der Aufbau von Datenfluß- und/oder Kontrollflußgraphen aus dem PROGRAMM. Diese Graphen können hinsichtlich ihrer möglichen Partitionierung und/oder Abbildung auf die Ziel-VPU automatisch untersucht werden. In diesem Fall werden die Teile der generierten Graphen bzw. die entsprechenden PROGRAMMTEILE, extrahiert, die sich hinreichend gut partitionieren und/oder abbilden lassen. Hierzu kann eine Partitionierbarkeits- und/oder Abbildbarkeitsanalyse erfolgen, die die jeweilige Eigenschaft bewertet Entsprechend dieser Bewertung erfolgt dann die Partitionierung und Extraktion der Programmteile auf die VPU, sowie das Einführen der vorgesehenen Interfaces. .

Es soll ausdrücklich auf die in der Patentanmeldung DE 101 39 170.6 beschriebenen Analysemethoden verwiesen werden, die beispielsweise zur Anwendung kommen können. Die vorerwähnte Anmeldung ist zu Offenlegungszweckung vollumfänglich eingegliedert.

Eine mögliche Analysemethode ist auch durch Erkennung bestimmter Datentypen gegeben.

Unterschiedliche Datentypen eignen sich mehr oder weniger gut für die Bearbeitung auf einer VPU. Beispielsweise ist eine komplexe Pointer-Arithmetik, bzw. eine pointerbasierende Datenadressierung (pointer) schwer auf eine VPU abbildbar, während sich Arrays (array) sehr gut abbilden lassen.

Erfindungsgemäß können daher weitgehend automatisch oder manuell die jeweils geeigneten Datentypen und zumindest wesentliche Teile von deren Datenverarbeitung auf eine VPU übertragen und entsprechend extrahiert werden. Die Extraktion erfolgt da-

mit im Ansprechen auf das Auftreten bestimmter Datentypen und/oder Datenoperationen.

Es soll erwähnt werden, daß zusätzliche, den Datentypen zugeordnete Parameter weitere Hinweise zur Bestimmung der Ausführbarkeit und/oder Ausführungsperformance auf einer VPU geben können und daher maßgeblich zur Extraktion mitverwendet werden können. Beispielsweise spielt die Größe von zu berechnenden Arrays eine wesentliche Rolle. Es lohnt sich zumeist nicht, kleine Arrays auf einer VPU zu berechnen, da hier der Synchronisations- und Datenaustauschaufwand zwischen CPU und VPU zu hoch sein kann. Einschränkend ist aber dabei wiederum zu erwähnen, daß kleine Arrays, die innerhalb einer Schleife besonders häufig verrechnet werden, sich dennoch sehr gut für VPUs eignen, insofern die Schleife weitestgehend komplett auf der VPU berechnet wird. Große Arrays können dagegen zumeist ohne weiteres auf einer VPU besonders performant berechnet werden.

Weiterhin soll erwähnt werden, daß besondere Datentypen durch einen besonders angepaßten Compiler oder ggf. durch einen Anwender (z. B. mittels TYPE in Pascal) erstellt werden können, die sich besonders für VPUs eignen und deren Datenverarbeitung dann auf einer VPU ausgeführt wird.
Beispielsweise können folgende Datentypen bestehen:

```
TYPE stream1 of Byte [];
TYPE stream2 of Byte [0..255;
```

Stream definiert einen Datenstrom (stream) von in der Regel großer, ggf. nicht vorbekannter und/oder unendlicher Länge. Stream1 hatte hier eine nicht vorbekannte Länge. Beispielsweise könnte ein mit diesem Datentyp programmierter FIR-Filter (oder z. B. eine FFT oder DCT) automatisch - und ggf ausgewalzt - auf eine VPU abgebildet werden. Die Rekonfiguration

erfolgt dann typisch und bevorzugt im Ansprechen auf andere Mechanismen als den Datenstromverlauf, z.b. durch Zähler, Vergleicher, CT-gesteuert und/oder durch Time-Out. Soll hierbei etwa eine Wave- oder andere Rekonfiguration ausgelöst werden, so kann diese über eine durch vorgenannte Methoden veranlaßte Kennzeichnung eines Datenpaketes, insbesondere Datenbytes, als ein letztes zu sein erfolgen um nach und/oder mit dem Durchlauf dieses als letzes Datenpaket gekennzeichneten Datenpaketes die Rekonfiguration auszulösen.

stream2 definiert einen Datenstrom der Länge von hier 256 Byte, der wie stream1 behandelt werden kann, jedoch die Eigenschaft aufweist, nach 256 Byte zu enden und damit nach Beendigung möglicherweise eine Rekonfiguration im Sinne der vorab zitierten Patente selbigen Anmelders auslösen kann. Insbesondere kann eine Wave-Rekonfiguration (z. B. nach DE 197 04 728.9, DE 199 26 538.0, DE 102 06 857.7, DE 100 28 397.7) mit dem Eintreffen des letzten Daten-Bytes ausgelöst werden und mit der Verarbeitung dieses letzten Daten-Bytes die jeweilige, das Byte verarbeitende PAE rekonfiguriert werden.

Eine für die implementierte VPU geeignete Übersetzung des extrahierten Codes nach NML kann bevorzugt durchgeführt werden.

Für datenflußorientierte VPUs kann beispielsweise automatisch ein Datenfluß- und/oder Kontrollflußgraph aufgebaut werden. Die Graphen werden dann in NML-Code übersetzt.

Entsprechende Code-Teile wie z. B. Schleifen können mittels einer Datenbank (LookUp) übersetzt werden oder gewöhnliche Transformationen können durchgeführt werden. Für Codeteile können auch Makros vorgesehen sein, die dann gemäß den in vorgenannten Anmeldungen offenbarten IKR weiterverwendet werden.

Ebenfalls kann die Modularisierung nach PACT13, Fig. 28 unterstützt werden.

Gegebenenfalls kann bereits das Abbilden auf die VPU bzw. dessen Vorbereitung erfolgen, beispielsweise mittels der Durchführung des Plazierens der benötigten Ressourcen und des Routens der Verbindungen (Place and Route). Dies kann zum Beispiel nach per se bekannten Regeln des Plazierens und Routens geschehen.

Es ist auch möglich, mittels einer automatischen Analysemethode den extrahierten Code und/oder den übersetzten NML-Code auf seine Verarbeitungseffizienz hin zu analysieren. Dabei ist die Analysemethode bevorzugt so gewählt, daß der Interface-Code und die daraus entstehenden Performanceeinflüsse an geeigneter Stelle mit in die Analyse einfließen. Geeignete Analyseverfahren sind insbesondere in den vorgenannten Anmeldungen der vorliegenden Anmelderin beschrieben.

Gegebenenfalls wird die Analyse durch eine komplette Übersetzung und Implementierung auf dem Hardware-System durchgeführt, indem das PROGRAMM ausgeführt und mit geeigneten Methoden, wie sie beispielsweise nach dem Stand der Technik bekannt sind, vermessen wird.

Es ist weiter möglich, daß basierend auf den durchgeführten Analysen, verschiedene durch die Extraktion für eine VPU gewählte Teile als ungeeignet identifiziert werden können. Umgekehrt kann die Analyse ergeben, daß bestimmte, für einen PROZESSOR extrahierte Teile zur Ausführung auf einer VPU geeignet wären.

Eine optionale Schleife, die nach der Analyse basierend auf geeigneten Entscheidungskriterien zurück in den Extraktionsteil führt, um diesen mit entsprechend der Analyse angepaßten Extraktionsvorgaben erneut auszuführen, ermöglicht die Optimierung des Übersetzungsergebnisses. Man hat somit eine Iteration. Dieses Vorgehen ist bevorzugt.

Eine Schleife kann an mehreren unterschiedlichen Stellen in den Compilerlauf eingebracht sein.

Der erhaltene NML-Code ist bei Bedarf entsprechend den Eigenschaften der verwendeten VPU zu partitionieren, d.h. in einzelne Teile zu zerlegen, die auf jeweils in die vorhandenen Ressourcen abgebildet werden können.

Eine Vielzahl derartiger Mechanismen, insbesondere auf Graphenanalyse basierende, sind nach dem Stand der Technik per se bekannt. Eine bevorzugte Variante basiert jedoch auf der Analyse der Programmsourcen und ist unter dem Begriff temporal Partitioning bekannt. Dieses Verfahren ist in der genannten PHD-Thesis von Cardoso beschrieben, die zu Offenbarungszwecken vollumfänglich eingegliedert wird.

Partitionierungsverfahren gleich welcher Art sind entsprechend des verwendeten VPU-Types zu adaptieren. Liegen VPUs vor, die die Speicherung von Zwischenergebnissen in Register und/oder Speicher zulassen, ist durch die Partitionierung die Einbindung der Speicher zur Speicherung von Daten und/oder Zuständen zu berücksichtigen. Die Partitionierungsalgorithmen (z. B. die temporale Partitionierung) sind entsprechend zu adaptieren. Gewöhnlicherweise wird die eigentliche Partitionierung und das Scheduling durch die genannten Patente jedoch erheblich vereinfacht und erst sinnvoll ermöglicht.

Manche VPUs bieten die Möglichkeit der differentiellen Rekonfiguration. Diese kann angewendet werden, wenn nur verhältnismäßig wenige Änderungen innerhalb der Anordnung der PAEs bei einer Rekonfiguration notwendig werden. Mit anderen Worten werden nur die Veränderungen einer Konfiguration gegenüber der aktuellen Konfiguration rekonfiguriert. Die Partitionierung kann in diesem Fall dergestalt sein, daß die auf eine Konfiguration folgende, gegebenenfalls differentielle Konfiguration nur die notwendigen Rekonfigurationsdaten enthält und keine vollständige Konfiguration darstellt. Es ist möglich, den Konfigurationsdatenoverhead zu Analysezwecken bei der Beurteilung der Aufteilungseffizient mit zu berücksichtigen.

Die Schedulingmechanismen für die partitionierten Codes können derart erweitert werden, daß das Scheduling durch Rückmeldungen der VPU an die jeweils rekonfigurierende Einheit (CT und/oder HOSTRECONF) gesteuert wird. Insbesondere wird dabei bei der Partitionierung die sich daraus ergebende Möglichkeit der bedingten Ausführung, d. h. der expliziten Bestimmung der nachfolgenden Partition durch den Zustand der aktuellen Partition genutzt. Mit anderen Worten ist es möglich, die Partitionierung derart zu optimieren, daß bedingte Ausführungen wie z. B. IF, CASE etc. berücksichtigt werden.

Werden VPUs verwendet, die die Fähigkeit besitzen Statussignale zwischen den PAEs zu übertragen, wobei PAEs auf die jeweils übertragenen Zustände reagieren und/oder diese mitverarbeiten, kann innerhalb der Partitionierung und des Schedulings zudem die bedingte Ausführung innerhalb der Anordnung der PAEs, also ohne die Notwendigkeit einer vollständigen oder teilweisen Rekonfiguration aufgrund eines geänderten bedingten Programmablaufs, berücksichtigt werden.

Weiterhin kann das Scheduling die Möglichkeit des Vorladens von Konfigurationen während der Laufzeit einer anderen Konfiguration unterstützen. Dabei können mehrere Konfigurationen möglicherweise auch spekulativ vorgeladen werden, d. h. ohne daß sichergestellt ist, daß die Konfigurationen überhaupt benötigt werden. Durch Selektionsmechanismen können dann zur Laufzeit die zu verwendenden Konfigurationen ausgewählt werden (siehe auch Beispiel NLS in DE 100 50 442.6, EP 01 102 674.7)

Eine zusätzliche oder alternative Variante sieht vor, dass die Datenverarbeitung innerhalb der an die CPU gekoppelten VPU exakt gleichviele Takte benötigt, wie die Datenverarbeitung innerhalb der Rechenpipeline der CPU. Insbesondere bei modernen Hochleistungs-CPUs mit einer Vielzahl von Pipelinestufen (>20) kann dieses Konzept ideal eingesetzt werden. Der besondere Vorteil ist, dass keine besonderen Synchronisationsmechanismen wie z.B. RDY/ACK notwendig sind und/oder keine Anpassung von Opcodes zur Registersteuerung erforderlich ist. Der Compiler hat bei diesem Verfahren sicherzustellen, dass die VPU die erforderliche Anzahl an Takten einhält und ggf. die Datenverarbeitung durch das Einfügen von Verzögerungsstufen wie z. B. einen Fall-Through FIFOs auszubalancieren wie er in anderen, vorerwähnten Anmeldungen beschrieben ist.

Der ausgegebene Code ist üblicherweise vollständig und bevorzugt ohne weitere Eingriffe auf den jeweils nachfolgenden Compilern verarbeitbar. Gegebenenfalls können Compilerflags und Constraints zur Steuerung nachfolgender Compiler generiert werden, wobei der Anwender falls gewünscht optional eigene Vorgaben hinzufügen und/oder die generierten Vorgaben modifizieren kann. Die nachfolgenden Compiler benötigen keine we-

sentlichen Modifikationen, so daß per se bekannte Standard-Tools prinzipiell einsetzbar sind.

Das vorgeschlagene Verfahren eignet sich somit beispielsweise insbesondere als Präprozessor bzw. Präprozessorverfahren vor Compilern und Entwicklungssystemen. Es soll aber ausdrücklich erwähnt werden, daß prinzipiell anstatt und/oder zusammen mit den zuvor beschriebenen Übersetzers auch Compiler nach PACT11 eingebunden werden können.

An die beschriebene Architektur, insbesondere direkt an die VPU kann ein FPGA gekoppelt sein, um feingranulare Datenverarbeitung zu ermöglichen und/oder ein flexibel adaptierbares Interface (z.B. diverse serielle Schnittstellen (V24, USB, etc.), diverse parallele Schnittstellen, Festplattenschnittstellen, Ethernet, Telekommunikationsschnittstellen (a/b, T0, ISDN, DSL, etc)) zu weiteren Baugruppen zu ermöglichen.
Der FPGA kann dabei aus der VPU-Architektur, insbesondere durch die CT, und/oder durch die CPU konfiguriert werden. Der FPGA kann statisch, also ohne Rekonfiguration zur Laufzeit und/oder dynamisch, also mit Rekonfiguration zur Laufzeit, betrieben werden.

Es wurde bereits das Vorsehen eines Interface-Code angesprochen. Der Interface-Code, der in den extrahierten Code eingesetzt wird, kann durch unterschiedliche Verfahren vorgegeben werden. Bevorzugt wird der Interface-Code in einer Datenbank abgelegt, auf die zugegriffen wird. Die Einheit zur Umsetzung kann so ausgebildet sein, daß sie eine Auswahl, etwa des Programmierers, berücksichtigt, bei der beispielsweise durch Hinweise im PROGRAMM oder durch Compilerflags der passende Interface-Code ausgewählt wird. Dabei kann ein für das jeweils ver-

wendete Implementierungsverfahren des VPU/CPU-Systems geeigneter Interface-Code gewählt werden..

Die Datenbank selbst kann durch unterschiedliche Methoden aufgebaut und gewartet werden. Einige Beispiele sollen zur Verdeutlichung der Möglichkeiten angeführt werden:

a) Der Interface-Code kann vom Lieferanten des Compilers für bestimmte Verbindungsverfahren zwischen VPU und CPU(s) vorgegeben werden. Dies kann bei der Organisation der Datenbank berücksichtigt werden, indem entsprechende Speichermittel für diese Angaben bereitgehalten werden.
b) Der Interface-Code kann vom Benutzer, der den Systemaufbau bestimmt hat, selbst geschrieben oder aus bestehenden (Beispiel-) Interface-Code modifiziert und der Datenbank zugefügt werden. Das Datenbankmittel wird hierzu bevorzugt benutzermodifizierbar gestaltet, um dem Benutzer die Datenbankmodifikation zu ermöglichen.
c) Der Interface-Code kann von eine Entwicklungssystem, mit dem beispielsweise der Systemaufbau des VPU-CPU-Systems geplant und/oder beschrieben und/oder getestet wurde, automatisch generiert werden.

Der Interface-Code ist gewöhnlicherweise bevorzugt derart gestaltet, daß er den Anforderungen der Programmiersprache entspricht, in der der extrahierte Code vorliegt in den der Interface-Code eingefügt werden soll.

Debugging und Integration der Toolsets

In die Interface-Codes können Kommunikationsroutinen eingeführt werden, um die unterschiedlichen Entwicklungssysteme für PROZESSOR und VPU zu synchronisieren. Insbesondere kann Code

für die jeweiligen Debugger (z. B. nach PACT11) aufgenommen werden.

Der Interface-Code ist so ausgebildet, daß er den Datenaustausch zwischen PROZESSOR und VPU ermöglicht und/oder steuert. Er ist daher eine geeignete und bevorzugte Schnittstelle, um die jeweiligen Entwicklungssysteme und Debugger zu steuern. Es ist beispielsweise möglich, einen Debugger für den PROZESSOR solange zu aktivieren, wie die Daten von dem Prozessor verarbeitet werden. Sobald die Daten über den Interface-Code an eine (oder mehrere) VPU übergeben werden, ist ein Debugger für VPUs zu aktivieren. Wird der Code zurück an den PROZESSOR gesendet, soll wiederum der PROZESSOR-Debugger aktiviert werden. Es ist daher also möglich und bevorzugt, derartige Abläufe durch das Einfügen von Steuercodes für Debugger und/oder Entwicklungssysteme in den Interface-Code abzuwickeln.

Die Kommunikation und Steuerung zwischen den unterschiedlichen Entwicklungssystemen soll daher bevorzugt mittels in die Interface-Codes von PROZESSOR und/oder VPU eingebrachte Steuercodes abgewickelt werden. Die Steuercodes können dabei bestehenden Standards für die Steuerung von Entwicklungssystemen weitgehend entsprechen.

Die Verwaltung und Kommunikation der Entwicklungssysteme wird vorzugsweise wie beschrieben in die Interface-Codes abgewikkelt, kann jedoch - sofern sinnvoll - auch getrennt von diesen, nach einem entsprechenden ähnlichen Verfahren abgewickelt werden.

In vielen Programmiersprachen, besonders in sequentiellen wie z. B. C, wird eine exakte zeitliche Reihenfolge implizit durch die Sprache vorgegeben. Bei sequentiellen Programmiersprachen

geschieht dies beispielsweise durch die Reihenfolge der einzelnen Anweisungen. Sofern durch Programmiersprache und/oder den Algorithmus erforderlich, läßt sich die Zeitinformation auf Synchronisationsmodelle wie RDY/ACK und/oder REQ/ACK oder ein Time-Stamp-Verfahren abbilden.

Beispielsweise wird eine nachfolgende for-Schleife nur dann durchlaufen und iteriert, wenn eine Variable, hier inputstream je Durchlauf mit einem RDY quittiert ist. Bleibt RDY aus, wird der Schleifendurchlauf bis zum Eintreffen RDY angehalten:

```
while TRUE
    s := 0
    for i: 1 to 3
        s := s + inputstream;
```

Die Eigenschaft der sequentiellen Sprachen, nur von der Befehlsverarbeitung gesteuert zu werden, wird mit dem Datenflußprinzip die Verarbeitung durch den Datenstrom, bzw. die Existenz von Daten zu steuern verbunden. Mit anderen Worten wird ein Befehl und/oder eine Anweisung (z. B. `s := s + inputstream;`) nur verarbeitet, wenn die Operation ausgeführt werden kann und die Daten verfügbar sind.

Bemerkenswert ist, daß dieses Verfahren gewöhnlicherweise zu keiner Änderung der Syntax oder Semantik einer Hochsprache führt.

Komplexere Funktionen einer Hochsprache, wie z. B. Schleifen, werden durch Makros realisiert. Die Makros werden vom Compiler vorgegeben und zur Übersetzungszeit instantiiert.

Die Makros sind entweder aus einfachen Sprachkonstrukten der Hochsprache oder auf Assemblerlevel aufgebaut. Makros können parametriert sein, um eine einfach Adaption an den beschriebenen Algorithmus zu ermöglichen. (vgl. auch PACT11)

Ein Standardprozessor z.B. ein RISC, CISC, DSP (CPU) wird also mit einem rekonfigurierbaren Prozessor (VPU) gekoppelt.

Zwei unterschiedliche, bevorzugt jedoch auch zugleich implementierbare Kopplungsvarianten könenn wie folgt beschrieben sein.

Eine erste Variante sieht eine direkte Ankoppelung an den Befehlssatz einer CPU vor (Befehlssatzkopplung).
Eine zweite Variante sieht eine Ankoppelung über Tabellen im Hauptspeicher vor. Es sind also Tabellenmittel vorgesehen.

Innerhalb eines Instruktionssatzes (ISA) einer CPU sind für gewöhnlich freie unbenutzte Befehle vorhanden. Einer oder eine Mehrzahl dieser freien unbenutzen Befehle wird nunmehr für die Steuerung von VPUs verwendet (VPUCODE).

Durch die Dekodierung eines VPUCODEs wird eine Konfigurationseinheit (CT) einer VPU angesteuert die in Abhängigkeit des VPUCODEs bestimmte Abläufe ausführt. Es ist also eine zur VPU-Decodierung ansprechbare CT vorhanden.

Beispielsweise kann ein VPUCODE das Laden und/oder Ausführen von Konfigurationen durch die Konfigurationseinheit (CT) für eine VPU auslösen.

In einer erweiterten Ausführung kann ein VPUCODE über eine Übersetzungstabelle, die bevorzugt von der CPU, alternativ aber auch von der oder einer VPU oder einer externen Einheit aus verwaltet wird, auf unterschiedliche VPU-Kommandos übersetzt werden

Die Konfigurationstabelle kann in Abhängigkeit von dem ausgeführten CPU Programm oder Codeabschnitt gesetzt werden.

Die VPU lädt nach Eintreffen eines Ladekommandos Konfigurationen aus einem eigenen oder mit der CPU geteilten Speicher. Insbesondere kann eine VPU-Konfiguration im Code des aktuell ausgeführten CPU-Programmes beinhaltet sein.

Nach Erhalt eines Ausführungskommandos führt eine VPU die auszuführende Konfiguration aus und die entsprechende Datenverarbeitung durch. Das Beenden der Datenverarbeitung kann durch ein Terminierungssignal (TERM) an die CPU angezeigt werden. Dazu sind entsprechende Signalleitungen/Interrupt-Eingänge usw. vorhanden und/oder ausgebildet.

Das Auftreten eines VPUCODEs können solange Wartezyklen auf der CPU ausgeführt werden, bis das Terminierungssignal (TERM) der Beendigung der Datenverarbeitung von der VPU eintrifft.

In einer bevorzugten Ausgestaltung wird mit der Verarbeitung der nächsten Codes fortgefahren. Tritt ein weiterer VPUCODE auf, kann sodann auf die Beendigung des vorhergehenden gewartet werden, oder sämtliche gestartete VPUCODEs werden in einer Verarbeitungspipeline eingereiht, oder ein Taskwechsel wird insbesondere wie nachfolgend beschrieben ausgeführt.
Die Beendigung einer Datenverarbeitung wird durch das Eintreffen des Terminierungssignal (TERM) in einem Statusregister si-

gnalisiert. Die Terminierungssignale treffen in der Reihenfolge einer möglichen Verarbeitungspipeline ein.

Die Datenverarbeitung auf der CPU kann durch das Testen des Statusregisters auf das Eintreffen eines Terminierungssignales synchronisiert werden.

In einer möglichen Ausgestaltung kann, sofern eine Applikation vor dem Eintreffen von TERM z.B. durch Datenabhängigkeiten nicht fortgesetzt werden kann, ein Taskwechsel ausgelöst werden.

Es ist bevorzugt, wenn lose Kopplungen zwischen Prozessoren und VPUs aufgebaut sind, bei welchen VPUs weitestgehend als unabhängige Coprozessoren arbeiten.

Eine derartige Kopplung sieht eine oder mehrere gemeinsame Datenquellen und -senken, zumeist über gemeinsame Bussysteme und/oder gemeinsame Speicher vor. Über DMAs und/oder andere Speicherzugriffskontroller werden Daten zwischen einer CPU und einer VPU ausgetauscht. Die Synchronisation der Datenverarbeitung erfolgt bevorzugt über eine Interruptsteuerung oder einen Statusabfragemechanismus (z.B. Polling).

Eine enge Ankopplung entspricht der vorab beschriebenen direkten Ankopplung einer VPU in den Befehlssatz einer CPU.

Bei einer direkten Rechenwerk-Ankopplung ist besonders auf eine hohe Rekonfigurationsperformance zu achten. Bevorzugt kann daher die Wave-Rekonfiguration zum Einsatz kommen. Desweiteren werden die Konfigurationsworte bevorzugt vorab derart vorgeladen, dass bei Ausführung des Befehls die Konfiguration beson-

ders schnell (mittels Wave-Reconfiguration im Optimalfall innerhalb eines Taktes) konfiguriert werden kann. Im übrigen wäre auch möglich, anstelle einer Array-Teilkonfiguration bei hoch performanten, insbesondere aber auch bei überwiegend niederperformanten Anwendungen mehrere insbesondere identische Arrays vorzusehen, von diesen wenigstens eines für eine neue Task umzukonfigurieren, insbesondere im Vorgriff, und dann nach Bedarf anstelle einer Umkonfiguration oder Teilumkonfiguration eines integralen multidimensionalen partiell zur Laufzeit rekonfigurierbaren grobgranularen Feldes einfach auf ein anderes Array vollständig zu wechseln. Signale können dabei z. B. über MUX-/Demuxstufen den Teilarrays zugeführt werden, insbesondere I/O-, Daten-, Status- und/oder Triggersignale.

Für die Wave-Reconfiguration werden bevorzugt die voraussichtlich auszuführenden Konfigurationen vorab durch den Compiler zur Compilezeit erkannt und zur Laufzeit entsprechend vorgeladen.

Zum Zeitpunkt der Befehlsausführung wird die entsprechende Konfiguration gegebenenfalls für jede PAE einzeln und/oder für eine PAE-Teilmenge einzeln selektiert und ausgeführt. Auch derartige Verfahren sind nach den o.g. Schriften bekannt.

Eine bevorzugte Implementierung kann unterschiedliche Datentransfers zwischen einer CPU und VPU vorsehen. Drei besonders bevorzugte einzeln oder kombiniert einsetzbare Methoden werden nachfolgend beschrieben.

Bei einer Registerkopplung kann die VPU Daten aus einem CPU-Register entnehmen, verarbeiten und in ein CPU-Register zurückschreiben.

Bevorzugt werden Synchronisationsmechanismen zwischen der CPU und der VPU eingesetzt.

Beispielsweise kann die VPU durch das Einschreiben der Daten in ein CPU-Register durch die CPU ein RDY-Signal erhalten und daraufhin die eingeschriebenen Daten verarbeiten. Das Auslesen von Daten aus einem CPU-Register durch die CPU kann ein ACK-Signal generieren, wodurch die Datenabnahme durch die CPU der VPU signalisiert wird. Die Verwendung des per se bekannten RDY/ACK-Protokolls in unterschiedlicher Ausprägung ist vorliegend gerade bei grobgranularen Zellen der rekonfigurierbaren Einheiten vorteilhaft.

CPUs stellen typischerweise keine entsprechenden Mechanismen zur Verfügung.

Zwei mögliche Lösungen werden näher beschrieben:

Ein einfach zu realsierenden Ansatz ist, die Datensynchronisation über ein Statusregister durchzuführen. Beispielsweise kann die VPU das erfolgte Auslesen von Daten aus einem Register und das damit verbundene ACK-Signal und/oder das Einschreiben von Daten in ein Register und das damit verbundene RDY-Signal in dem Statusregister anzeigen. Die CPU testet zunächst das Statusregister und führt beispielsweise so lange Warteschleifen oder Taskwechsel aus, bis - je nach Operation - das RDY oder ACK eintraf. Danach führt die CPU den jeweiligen Registerdatentransfer aus.

In einer erweiterten Ausgestaltung wird der Befehlssatz der CPU um load/store-Instruktionen mit integrierter Statusabfrage (load_rdy, store_ack) erweitert. Beispielsweise wird bei einem store_ack nur dann ein neues Datenwort in ein CPU-Register ge-

schrieben, wenn das Register vorher von der VPU ausgelesen wurde und ein ACK eintraf. Entsprechend liest load_rdy nur Daten aus einem CPU-Register, wenn die VPU vorher neue Daten eingeschrieben und ein RDY generiert hat.

Daten, die zu einer auszuführenden Konfiguration gehören können sukzessive, quasi durch Block-Moves ähnlich wie nach dem Stand der Technik in die CPU-Register geschrieben und/oder aus diesen gelesen werden. Ggf. implementierte Block-Move-Instruktionen können bevorzugt durch die beschriebene integrierte RDY/ACK Statusabfrage erweitert werden.

Es ist offensichtlich, dass eine Vielzahl von leichten Modifikationen und unterschiedlichen Ausgestaltungen dieses Grundverfahrens möglich sind.

Die bereits erwähnte Wave-Rekonfiguration erlaubt das Starten eines neuen VPU-Befehls und der entsprechenden Konfiguration, sobald die Operanden des vorhergehenden VPU-Befehls aus den CPU-Registern abgenommen wurden. Die Operanden für den neuen Befehl können direkt nach Befehlsstart in die CPU-Register geschrieben werden.

Entsprechend des Wave-Rekonfiguration-Verfahrens wird die VPU successive mit Fertigstellung der Datenverarbeitung des vorherigen VPU-Befehls für den neuen VPU-Befehl umkonfiguriert und die neuen Operanden verarbeitet.

Weiterhin können Daten zwischen einer VPU und einer CPU durch geeignete Buszugriffe auf gemeinsame Ressourcen ausgetauscht werden.

Sofern Daten ausgetauscht werden sollen, die kurz zuvor von der CPU verarbeitet wurden und daher voraussichtlich noch im bevorzugt vorzusehenden Cache der CPU liegen bzw. sofort anschliessend von der CPU verarbeitet werden und daher sinnvollerweise in den Cache der CPU gelegt werden, werden diese bevorzugt von der VPU aus dem Cache der CPU gelesen, bzw. in den Cache der CPU geschrieben. Dies kann durch geeignete Analysen weitestgehend vorab zur Compilezeit der Applikation durch den Compiler festgestellt und der Binärcode entsprechend generiert werden.

Sofern Daten ausgetauscht werden sollen, die sich voraussichtlich nicht im Cache der CPU befinden bzw. voraussichtlich nicht nachfolgend im Cache der CPU benötigt werden, werden diese bevorzugt von der VPU direkt vom externen Bus und der damit verbundenen Datenquelle (z.B. Speicher, Peripherie) gelesen, bzw. an den externen Bus und der damit verbundenen Datensenke (z.B. Speicher, Peripherie) geschrieben. Dies kann durch geeignete Analysen weitestgehend vorab zur Compilezeit der Applikation durch den Compiler festgestellt und der Binärcode entsprechend generiert werden.

Bei einem Transfer über den Bus am Cache vorbei wird bevorzugt ein Protokoll zwischen Cache und Bus implementiert, das für einen korrekten Inhalt des Caches sorgt. Beispielsweise kann das bekannte MESI-Protokoll nach dem Stand der Technik hierzu verwendet werden.

Die beschriebenen Verfahren müssen zunächst keinen besonderen Mechanismus für die Unterstützung von Betriebssystemen vorsehen. Es ist nämlich bevorzugt, sicherzustellen, dass ein auszuführendes Betriebssystem sich entsprechend des Status einer

zu unterstützenden VPU verhält, was möglich ist und wozu insbesondere Scheduler vorgesehen sein können.

Bei einer engen Rechenwerkkopplung wird bevorzugt das Statusregister der CPU abgefragt, in welches die angekoppelte VPU ihren Datenverarbeitungsstatus (Terminierungssignal) einträgt. Soll eine weitere Datenverarbeitung an die VPU übertragen werden, und die VPU hat die vorherige Datenverarbeitung noch nicht beendet wird gewartet und/oder bevorzugt ein Taskwechsel ausgeführt.

Für eine Coprozessorkopplung werden bevorzugt über das Betriebssystem, i.b. den Scheduler gesteuerte Mechanismen verwendet:

Ein einfacher Scheduler kann nach Übertragung einer Funktion auf eine VPU entweder den aktuellen Task auf der CPU weiterlaufen lassen, sofern dieser unabhängig und parallel zur Datenverarbeitung auf einer VPU ablaufen kann. Sofern oder sobald der Task auf die Beendigung der Datenverarbeitung auf der VPU warten muss, schaltet der Taskscheduler auf einen anderen Task um.

Jeder neu aktivierte Task wird, sofern er die VPU verwendet, vor Verwendung prüfen, ob diese für eine Datenverarbeitung zur Verfügung steht und/oder aktuell noch Daten verarbeitet; dann soll entweder auf die Beendigung der Datenverarbeitung gewartet oder bevorzugt der Task gewechselt werden.

Ein einfaches und dennoch leistungsfähiges Verfahren kann durch sogenannte Descriptor Tables aufgebaut werden, die beispielsweise folgendermaßen realisiert werden können:

Jeder Task generiert zum Aufruf der VPU eine oder mehrere Tabelle(n) (VPUCALL) mit einem geeigneten festgelegten Datenformat in dem ihm zugewiesenen Speicherbereich. Diese Tabelle beinhaltet sämtliche Steuerinformation für eine VPU, wie z.B. das auszuführende Programm / die auszuführende Konfiguration und/oder Zeiger auf die Speicherstelle(n) oder Datenquellen der Eingangsdaten und/oder die Speicherstelle(n) oder Datensenken der Ergebnisdaten und/oder weitere Ausführungsparameter, z.B. Datenarraygrößen.

Im Speicherbereich des Betriebssystems befindet sich eine Tabelle oder verkettete Liste (LINKLIST), die auf sämtliche VPUCALL-Tabellen in der Reihenfolge ihrer Erstellung zeigt.

Die Datenverarbeitung auf der VPU läuft nunmehr derart ab, dass ein Task einen VPUCALL erstellt und über das Betriebssystem die VPU aufruft. Das Betriebssystem erstellt einen Eintrag in der LINKLIST. Die VPU arbeitet die LINKLIST ab und führt die jeweils referenzierten VPUCALL aus. Die Beendigung einer der jeweiligen Datenabarbeitung wird jeweils durch einen entsprechenden Eintrag in die LINKLIST und/oder VPUCALL Tabelle angezeigt.

Die VPU arbeitet somit weitgehend unabhängig von der CPU. Das Betriebssystem und/oder die jeweiligen Task müssen lediglich die Tabellen (LINKLIST bzw. VPUCALL) überwachen.

Besonders performanceeffizient arbeiten die beiden Verfahren, wenn als VPU eine Architektur zum Einsatz kommt, die eine mit der Datenverarbeitung überlagerte und/oder überlagerbare Rekonfiguration zulässt.

Damit ist es möglich, eine neue Datenverarbeitung und eine ggf. damit verbundene Rekonfiguration sofort nach Lesen der letzten Operanden aus den Datenquellen zu starten. Mit anderen Worten ist für die Synchronisation nicht mehr das Beenden der Datenverarbeitung, sondern das Lesen der letzten Operanden erforderlich. Dadurch wird die Performance der Datenverarbeitung erheblich gesteigert.

inen zusätzlichen Einfluß auf die Betrachtung und den Umgang mit Zuständen hat der mögliche Einsatz eines Betriebssystemes. Betriebssysteme verwenden beispielsweise Task-Scheduler zum Verwalten mehrere Aufgaben (Tasks), um ein Multitasking zur Verfügung zu stellen.
Task-Scheduler brechen Tasks zu einem bestimmten Zeitpunkt ab, starten andere Tasks und kehren nach deren Abarbeitung zur Weiterbearbeitung des abgebrochenen Tasks zurück.
Sofern sichergestellt ist, daß eine Konfiguration - die der Abarbeitung eines Tasks entspricht - nur nach der kompletten Abarbeitung - d.h. wenn alle innerhalb dieses Konfigurationszyklusses zu bearbeitende Daten und Zustände gespeichert sind - terminiert, können lokal relevante Zustände ungespeichert bleiben.
Sofern der Task-Scheduler allerdings Konfigurationen vor deren vollständiger Abarbeitung abbricht, müssen lokale Zustände und/oder Daten gespeichert werden. Weiterhin ist dies von Vorteil, wenn die Abarbeitungszeit einer Konfiguration nicht vorhergesagt werden kann. In Verbindung mit dem bekannten Halteproblem und dem Risiko, daß eine Konfiguration (z.B. durch einen Fehler) gar nicht terminiert, erscheint dies weiterhin sinnvoll, um damit einen Deadlock des gesamten Systems zu verhindern.
Mit anderen Worten sind, unter Berücksichtigung von Taskwechseln, relevante Zustände auch als solche anzusehen, die für

einen Taskwechsel und ein erneutes korrekes Aufsetzen der Datenverarbeitung notwendig sind.
Bei einem Taskswitch ist somit der Speicher für Ergebnisse und ggf. auch der Speicher für die Operanden zu sichern und zu einem späteren Zeitpunkt, also bei der Rückkehr zu diesem Task, wieder herzustellen. Dies kann vergleichbar zu den PUSH/POP Befehlen und Verfahren nach dem Stand der Technik erfolgen. Weiterhin ist der Zustand der Datenverarbeitung zu sichern, also der Zeiger auf die zuletzt vollständig bearbeiteten Operanden. Es sei hier besonders auf PACT18 verwiesen.

Abhängig von der Optimierung des Taskswitches gibt es beispielsweise zwei Möglichkeiten:
a) Die abgebrochene Konfiguration wird neu konfiguriert und nur die Operanden werden geladen. Die Datenverarbeitung beginnt von neuem, als ob die Bearbeutung der Konfiguration noch gar nicht begonnen wurde. Mit anderen Worten werden einfach alle Datenberechnungen von vorne an ausgeführt, wobei ggf. Berechnungen bereits zuvor durchgeführt wurden. Diese Möglichkeit ist einfach aber nicht sehr effizient.
b) Die abgebrochene Konfiguration wird neu konfiguriert, wobei die Operaden und bereits berechneten Ergebnisse in die jeweiligen Speicher geladen werden. Die Datenverarbeitung wird bei den Operanden fortgesetzt die nicht mehr vollständig berechnet wurden. Dieses Verfahren ist sehr viel effizienter, setzt aber voraus, daß ggf. zusätzliche Zustände die während der Verarbeitung der Konfiguration entstehen relevant werden, beispielsweise muß zumindest ein Zeiger auf die zuletzt vollständig verechneten Operanden gesichert werden, damit bei deren Nachfolgern nach erfolgter neuer Konfiguration neu aufgesetzt werden kann.

Eine besonders bevorzugte Variante zur Verwaltung von relevanten Daten wird durch den nachfolgend beschriebenen Kontext Switch zur Verfügung gestellt. Bei Task-Wechseln und/oder bei

der Ausführung von Konfigurationen und derem Wechsel (siehe beispielsweise Patentanmeldung PACT15, die zu Offenbarungszwecken vollumfänglich eingegliedert ist) kann es erforderlich sein, Daten oder Zustände, die typischerweise nicht zusammen mit den Arbeitsdaten in die Speicher abgelegt werden, da sie beispielsweise lediglich einen Endwert markieren, für eine nachfolgende Konfiguration zusichern.

Der erfindungsgemäße Kontext Switch wird derart durchgeführt, dass eine erste Konfiguration entfernt wird, die zu sichernden Daten verbleiben in den entsprechenden Speichern (REG) (Speicher, Register, Zähler, etc).
Eine zweite Konfiguration wird geladen, diese verbindet die REG in geeigneter Weise und definierter Reihenfolge mit einem oder mehreren globalen Speicher(n).
Die Konfiguration kann beispielsweise Adressgeneratoren verwenden um auf den/die globalen Speicher zuzugreifen.
Die Konfiguration kann beispielsweise Adressgeneratoren verwenden um auf als Speicher ausgestaltete REG zuzugreifen.
Entsprechend der konfigurierten Verbindung zwischen den REG werden die Inhalte der REG in einer definierten Reihenfolge in den globalen Speicher geschrieben, wobei die jeweiligen Adressen von Adressgeneratoren vorgegeben werden. Der Adressgenerator generiert die Adressen für den/die globalen Speicher(n) derart, dass die beschriebenen Speicherbereiche (PUSHAREA) der entfernten ersten Konfiguration eindeutig zugeordnet werden können.
Mit anderen Worten, es sind bevorzugt für unterschiedliche Konfigurationen unterschiedliche Adressenräume vorgesehen. Die Konfiguration entspricht einem PUSH gewöhnlicher Prozessoren.

Danach verwenden andere Konfigurationen die Ressourcen.

Die erste Konfiguration soll wieder gestartet werden. Zuvor wird eine dritte Konfiguration gestartet, die die REG der er-

sten Konfiguration in einer definierten Reihenfolge miteinander verbindet.

Die Konfiguration kann beispielsweise Adressgeneratoren verwenden um auf den/die globalen Speicher zuzugreifen.

Die Konfiguration kann beispielsweise Adressgeneratoren verwenden um auf als Speicher ausgestaltete REG zuzugreifen.

Ein Adressgenerator generiert Adressen derart, dass ein korrekter Zugriff auf die der ersten Konfiguration zugeordnete PUSHAREA erfolgt. Die generierten Adressen und die konfigurierte Reihenfolge der REG sind derart, dass die Daten der REG in der ursprünglichen Ordnung aus den Speichern in die REG geschrieben werden. Die Konfiguration entspricht einem POP gewöhnlicher Prozessoren.

Die erste Konfiguration wird wieder gestartet.

Zusammengefaßt wird ein Kontext Switch derart durchgeführt, dass durch das Laden besonderer Konfigurationen, die ähnlich von PUSH/POP bekannter Prozessorarchitekturen arbeiten, die zu sichernden Daten mit einem globalen Speicher ausgetauschen werden.

Die Funktion soll in einem Beispiel verdeutlicht werden:
Eine Funktion addiert 2 Zahlenreihen, die Länge der Reihen ist zur Übersetzungszeit nicht bekannt, sondern erst zur Laufzeit.

```
proc example
  while i<length do
    x[i] = a[i] + b[i]
```

Die Funktion wird nun während ihrer Ausführung unterbrochen, beispielsweise durch einen Task-Switch oder weil der für x

vorgesehene Speicher voll ist. a,b,x befinden sich zu diesem Zeitpunkt erfindungsgemäß in Speichern. i und ggf. length müssen jedoch gesichert werden.
Dazu wird die Konfiguration example terminiert, wobei die Registerinhalte erhalten bleiben und eine Konfiguration push gestartet, die i und length aus den Registern liest und in einen Speicher schreibt.

```
proc push
  mem[<push_adr_example>] = i
  push_adr_example++
  mem[<push_adr_example>] = length
```

Nach der Ausführung wird push terminiert und die Registerinhalte können gelöscht werden.
Andere Konfigurationen werden ausgeführt. Nach einiger Zeit wird die Konfiguration example wieder gestartet.
Zuvor wird eine Konfiguration pop gestartet, die die Registerinhalte wieder aus dem Speicher liest.

```
proc pop
  i = mem[<push_adr_example>]
  push_adr_example++
  length = mem[<push_adr_example>]
```

Nach der Ausführung wird pop terminiert und die Registerinhalte bleiben bestehen. Die Konfiguration example wird wieder gestartet

Beschreibung der Figuren

Figur 1 verdeutlicht ein Beispiel das vorgeschlagene Verfahren und zeigt einen möglichen Systemaufbau. Dabei ist ein PROZES-

SOR (0101) über ein geeignetes Interface (0102) zum Daten- und Status-austausch mit einer VPU (0103) verbunden.
Ein PROGRAMM-Code (0110) wird (z. B. durch einen Präprozessor für einen Compiler) beispielsweise gemäß den beschriebenen Extraktionsmethoden in einen für den PROZESSOR geeigneten Teil (0111) und einen VPU-geeigneten Teil (0112) zerlegt.

0111 wird durch einen dem PROGRAMM-Code entsprechenden Standard Compiler (0113) übersetzt, wobei zuvor der zusätzliche Code zur Beschreibung und Verwaltung des Interfaces (0102) zwischen dem PROZESSOR und einer VPU aus einer Datenbank (0114) eingefügt wird. Auf 0101 ausführbarer sequentieller Code wird generiert (0116) und sofern notwendig die entsprechende Programmierung (0117) des Interfaces (0102).
Der Standard-Compiler kann dergestalt sein, daß er als marktübliches Werkzeug oder im Rahmen einer marktüblichen Entwicklungsumgebung vorliegt. Der Präprozessor und/oder möglicherweise der VPU-Compiler und/oder möglicherweise der Debugger und weitere Werkzeuge können beispielsweise in eine bestehende marktübliche Entwicklungsumgebung integriert werden.

0112 wird durch einen VPU Compiler (0115) übersetzt, wobei zusätzlicher Code zur Beschreibung und Verwaltung des Interfaces (0102) aus einer Datenbank (0114) eingefügt wird. Auf 0103 ausführbare Konfigurationen werden generiert (0118) und sofern notwendig die entsprechende Programmierung (0119) des Interfaces (0102). Es soll ausdrücklich erwähnt werden, daß prinzipiell auch Compiler nach DE 101 39 170.6 für 0115 verwendet werden können.

In Figur 2 ist beispielhaft ein prinzipieller Ablauf einer Compilation dargestellt. Ein PROGRAMM (0201) wird in der Extraktionseinheit (0202) nach unterschiedlichen Verfahren in

VPU-Code(0203) und PROZESSOR-Code (0204) zerlegt. Unterschiedliche Methoden können in beliebiger Kombination zur Extraktion angewendet werden, beispielsweise Hinweise im ursprünglichen PROGRAMM (0205) und/oder Unterprogrammaufrufe (0206) und/oder Analyseverfahren (0207) und/oder eine Verwertung von objektorientierten Klassenbibliotheken (0206a). Der jeweils extrahierte Code wird ggf. übersetzt und ggf. auf seine Eignung für das jeweilige Zielsystem hin überprüft (0208). Dabei ist eine Rückkopplung (0209) auf die Extraktion möglich, um Verbesserungen durch eine geänderte Zuordnung der Codes zu einem PROZESSOR oder einer VPU bzw. einer Vielzahl derselben zu erhalten.
Danach (0211) wird 0203 durch den Interface-Code aus einer Datenbank (0210) erweitert (0212) und/oder 0204 wird durch den Interface-Code aus 0210 zu 0213 erweitert.
Der entstandene Code wird auf seine Performance analysiert (0214), ggf. ist eine Rückkopplung (0215) auf die Extraktion möglich, um Verbesserungen durch eine geänderte Zuordnung der Codes zum PROZESSOR oder einer VPU zu erhalten.
Der entstandene VPU-Code (0216) wird für eine weitere Übersetzung an einen nachgeschalteten für die VPU geeigneten Compiler weitergegeben. Der entstandene PROZESSOR-Code (0217) wird für die weitere Übersetzung in einem beliebigen nachgeschalteten für den PROZESSOR geeigneten Compiler weiterverarbeitet.

Es soll angemerkt werden, daß einzelne Schritte je nach Verfahren ausgelassen werden können. Wesentlich ist, daß ein zumindest weitgehend kompletter und ohne, wenigstens ohne signifikanten Eingriff durch den Programmierer direkt übersetzbarer Code an jeweils nachgeschaltete Compilersysteme ausgegeben wird.

Es wird demnach vorgeschlagen, daß ein Präprozessormittel mit einem Codeeingang für die Einspeisung von zu compilierendem Code, mit Codeanalysemitteln, insbesondere Codestruktur und/oder Datenformat- und/oder Datenstroms-Erkennungs- und/oder Bewertungsmitteln sowie mit einem Aufteilungsbewertungsmittel zur Bewertung einer im Ansprechen auf Signale aus dem Codeanalysemittel vorgenommenen Codeaufteilung sowie gegebenenfalls einem Iterationsmittel zur Wiederholung einer Codeaufteilung bis zum Erreichen stabiler und/oder hinreichend akzeptabler Werte mit zumindest zwei Teilcodeausgängen versehen ist, wobei ein erster Teilcodeausgang Teilcode für zumindest einen herkömmlichen Prozessor ausgibt, und wenigstens ein weiterer Teilcodeausgang zur Abarbeitung mit rekonfigurierbaren Logikeinheiten, insbesondere mehr- bzw. multidimensionale insbesondere Zellstrukturen aufweisend, insbesondere grobgranulare datenverarbeitende und/oder Logikzellen (PAEs) mit Rechenwerken und dergleichen sowie ggf. zugeordneten Registermitteln und/oder feingranularen Steuer- und/oder Kontrollmitteln wie Zustandsmaschinen, RDY/ACK-Trigger- und Kommunikationsleitungen usw bestimmten Code ausgibt. Beide Teilcodeausgänge können in multiplexweise seriell auf einem physikalischen Ausgang liegen.

Die Datenbank für die Interface-Codes (0210) wird unabhängig und vor dem Compilerdurchlauf aufgebaut. Beispielsweise sind folgende Quellen für die Datenbank möglich: Vom Lieferanten vorgegeben (0220), vom Benutzer programmiert (0221) oder automatisch von einem Entwicklungssystem generiert (0222).

Der Aufbau einer besonders bevorzugten VPU ist in Figur 3 dargestellt. Vorzugsweise hierarchische Konfigurationsmanager (CT's) (0301) steuern und verwalten eine Anordnung von rekonfigurierbaren Elementen (PACs) (0302). Den CT's ist ein loka-

ler Speicher für die Konfigurationen zugeordnet (0303). Der Speicher verfügt weiterhin über ein Interface (0304) zu einem globalen Speicher, der die Konfigurationsdaten zur Verfügung stellt. Über ein Interface (0305) sind die Konfigurationsabläuft steuerbar. Ein Interface der rekonfigurierbaren Elemente (0302) zur Ablaufsteuerung und Ereignisverwaltung (0306) ist vorhanden, ebenso ein Interface zum Datenaustausch (0307).

Figur 4 zeigt einen Ausschnitt aus einem beispielhaften CPU System, beispielsweise einem DSP des Typs C6000 von Texas Instruments (0401). Dargestellt sind Programmspeicher (0402), Datenspeicher (0403), beliebige Peripherie (0404) und EMIF (0405). Über einen Speicherbus (0406) und einem Peripheriebus (0407) ist eine VPU als Coprozessor integriert (0408). Ein DMA-Kontroller (EDMA) (0409) kann beliebige DMA-Transfers, beispielsweise zwischen Speicher (0403) und VPU (0408) oder Speicher (0403) und Peripherie (0404) durchführen.

Figur 5 zeigt eine abstraktere Systemdefinition. Einer CPU (0501) ist Speicher (0502) zugeordnet auf den diese schreibenden und/oder lesenden Zugriff besitzt. Eine VPU (0503) ist mit dem Speicher gekoppelt. Die VPU ist in einen CT-Teil (0509) und die rekonfigurierbaren Elemente zur Datenverarbeitung (0510) untergliedert.

Zur Steigerung der Speicherzugriffe kann der Speicher mehrere unabhängige Zugriffsbusse aufweisen (multiport). In einer besonders bevorzugten Ausgestaltung ist der Speicher in mehrere unabhängige Segmente (Speicherbanks) segmentiert, wobei auf jede Bank unabhängig zugriffen werden kann. Sämtliche Segmente liegen vorzugsweise innerhalb eines einheitlichen Adressraums.

Vorzugsweise steht ein Segment hauptsächlich für die CPU zur Verfügung (0504), ein weiteres Segment steht hauptsächlich für die Datenverarbeitung der VPU zur Verfügung (0505), ein weiteres Segment steht hauptsächlich für die Konfigurationsdaten der VPU zur Verfügung (0506).
Typischerweise und bevorzugt weist eine vollausgestaltete VPU eigene Adressgeneratoren und/oder DMAs auf um Datentransfers durchzuführen. Alternativ und/oder zusätzlich ist es möglich, dass ein DMA (0507) innerhalb des Systems (Fig.5) für Datentransfers mit der VPU vorgesehen ist.

Das System enthält IO (0508) auf die CPU und VPU Zugriff haben können.

Sowohl CPU als auch VPU können jeweils dedizierte Speicherbereiche und IO-Bereiche aufweisen, auf die der jeweils andere keinen Zugriff hat.

Ein Datensatz (0511) der im Speicherbereich und/oder im IO-Bereich und/oder partiell in einem von beiden liegen kann wird zur Kommunikation zwischen CPU und VPU verwendet, z.B. zum Austausch von Basisparametern und Steuerinformation. Der Datensatz kann beispielsweise folgende Information beeinhalten:

1. Basisadresse(n) des CT-Speicherbereiches in 0506 zur Lokalisierung der Konfigurationen.
2. Basisadresse(n) von Datentransfers mit 0505.
3. IO Adressen von Datentransfers mit 0508.
4. Synchronisationsinformation, z.B. Zurücksetzen, anhalten, starten der VPU.
5. Statusinformation der VPU, z.B. Fehler oder Zustand der Datenverarbeitung.

Die Synchronisation der CPU und VPU erfolgt durch Polling von Daten und/oder bevorzugt durch Interruptsteuerung (0512).

Figur 6 zeigt eine mögliche Ausgestaltung der Interfacestruktur einer VPU zur Einbindung in ein System ähnlich Figur 5. Dazu werden der VPU ein Speicher/DMA- und/oder IO-Interface zum Datentransfer zugeordnet (0601), ein weiteres System-Interface (0602) übernimmt die Ablaufsteuerung wie z.B. das Verwalten von Interrupts, das Starten/Stoppen der Verarbeitung, Austausch von Fehlerzuständen, etc..
Das Speicher/DMA- und/oder IO-Interface wird an einen Speicherbus und/oder IO-Bus angeschlossen.

Das System-Interface wird vorzugsweise an einen IO-Bus angeschlossen, kann jedoch alternativ oder zusätzlich entsprechend 0511 auch an einen Speicher angeschlossen sein.

Die Interfaces (0601, 0402) können zur Anpassung von unterschiedlichen Arbeitsfrequenzen von CPU und/oder VPU und/oder System ausgestaltet sein, beispielsweise kann das System bzw. die CPU mit zB derzeit 500MHz und die VPU mit 200MHz arbeiten.

Die Interfaces können eine Übersetzung der Busprotokolle durchführen, beispielsweise kann das VPU interne Protokoll auf ein externes AMBA-Busprotokoll umgesetzt werden. Sie bewirken also Busprotokollübersetzungsmittel und/oder sind für die Busprotokollübersetzung ausgebildet, insbesondere die Busprotokollübersetzung zwischen internem VPU-Protokoll und bekanntem Busprotokoll. Es ist auch möglich, eine Konvertierung direkt auf CPU-interne Busprotokolle vorzusehen.

Das Speicher/DMA- und/oder IO-Interface unterstützt den Speicherzugriff der CT auf einen externen Speicher, der vorzugs-

weise direkt (memory mapped) erfolgt. Der Datentransfer der CT(s) und/oder PAC(s) kann gepuffert z.B. über FIFO-Stufen erfolgen. Externer Speicher kann direkt angesprochen und adressiert werden, weiterhin können DMA interne und/oder externe DMA-Transfers durchgeführt werden.
Über das System-Interface erfolgt die Steuerung der Datenverarbeitung, wie beispielsweise die Initialisierung und/oder der Start von Konfigurationen. Des weiteren werden Status und/oder Fehlerzustände ausgetauscht. Interrupts für die Steuerung und Synchronisation zwischen den CT's und einer CPU können unterstützt werden.

Das System-Interface kann VPU-interne Protokolle derart konvertieren, dass diese auf externe (Standard)-Protokolle umgesetzt werden (z.B. AMBA).

Ein bevorzugtes Verfahren zur Codegenerierung für das beschriebene System ist in anderen Teilen dieser Anmeldung beschrieben. Das Verfahren beschreibt einen Compiler, der Programmcode in Code für eine CPU und Code für eine VPU zerteilt. Nach unterschiedlichen Verfahren wird die Zerlegung auf die unterschiedlichen Prozessoren durchgeführt. In einer besonders bevorzugten Ausführung werden dabei die jeweiligen zerlegten Codes um die Interface-Routinen zur Kommunikation zwischen CPU und VPU erweitert. Die Erweiterung kann automatisch durch den Compiler erfolgen.

Die nachfolgende Tabellen zeigen beispielhafte Kommunikationen zwischen einer CPU und einer VPU. Den Spalten sind die jeweilig aktiven Funktionseinheiten zugeordnet: CPU, System-DMA und DMA-Interface (EDMA) bzw. Speicher-Interface (Speicher-I/F), System-Interface (System-I/F, 0602), CT's, sowie die PAC. In den Zeilen sind die einzelnen Zyklen in ihrer Ausführungsrei-

henfolge eingetragen. K1 referenziert eine auszuführende Konfiguration 1.

Die erste Tabelle zeigt beispielsweise einen Ablauf bei Verwendung der System-DMA (EDMA) zum Datentransfer:

CPU	EDMA	System-I/F	CT's	PAC
Initiiere K1				
	Lade K1			
Starte K1			Konfiguriere K1	
Initiiere laden der Daten per EDMA		Starte K1		Warten auf Daten
Initiiere lesen der Daten per EDMA	Datentransfer lesen der Daten			Datenverarbeitung
	Datentransfer schreiben der Daten	Signalisiere Ende der Operation		

Es ist zu erwähnen, dass die Synchronisation zwischen der EDMA und der VPU automatisch über das Interface 0401 erfolgt, d.h. DMA-Tranfers finden nur statt, wenn die VPU dafür bereit ist.

In einer zweiten Tabelle ist beispielsweise ein bervorzugter optimierter Ablauf dargestellt. Die VPU besitzt selbst direkten Zugriff auf den Konfigurationsspeicher (0306). Desweiteren werden die Datentransfers durch DMA-Schaltung innerhalb der VPU ausgeführt, die beispielsweise fest implementiert sein

können und/oder durch die Konfiguration von konfigurierbaren Teilen der PAC entstehen.

CPU	EDMA	System-I/F	CT's	PAC
Initiiere K1				
Starte K1	Lesen der Konfiguration		Konfiguriere K1	
	Datentransfer lesen der Daten	Starte K1		Lese Daten
				Datenverarbeitung
	Datentransfer schreiben der Daten	Signalisiere Ende der Operation		Schreibe Daten

Der Aufwand für die CPU ist minimal.

Zusammenfassend befaßt sich die vorliegende Erfindung mit Verfahren, die eine Übersetzung einer klassischen Hochsprache wie Pascal, C, C++, Java, etc. auf eine rekonfigurierbare Architektur ermöglicht. Das Verfahren ist derart ausgelegt, daß nur die jeweils für die rekonfigurierbare Zielarchitektur geeigneten Teile des zu übersetzenden Programmes extrahiert werden. Die verbleibenden Teile des Programmes werden auf eine konventionelle Prozessorarchitektur übersetzt.

In Figur 7 sind aus Gründen der Übersichtlichkeit nur die relevanten Komponenten (i.b. der CPU) aufgezeigt sind, wobei ty-

pisch eine wesentliche Zahl weiterer Komponenten und Netzwerke vorhanden sein wird.

Eine bevorzugte Implementierung wie beispielsweise in Figur 1 dargestellt kann unterschiedliche Datentransfers zwischen einer CPU (0701) und VPU (0702) vorsehen. Die auf der VPU auszuführenden Konfigurationen werden durch den Instruktionsdekoder (0705) der CPU selektiert, der bestimmte für die VPU bestimmte Instruktionen erkennt und die CT (0706) derart ansteuert, dass diese die entsprechenden Konfigurationen aus einem der CT zugeordneten Speicher (0707) - der insbesondere mit der CPU geshared werden oder derselbe wie der Arbeitsspeicher der CPU sein kann, in das Array aus PAEs (PA, 0108) lädt.

Es sind CPU-Register (0703) vorgesehen, um bei einer Registerkopplung Daten zu entnehmen, zu verarbeiten und in ein CPU-Register zurückschreiben. b Für die Datensynchronisation ist ein Statusregister (0704) vorgesehen. Weiter ist ein Cache vorgesehen, der dafür vorgesehren ist, daß wenn Daten ausgetauscht werden sollen, die kurz zuvor von der CPU verarbeitet wurden, diese voraussichtlich noch im Cache (0709) der CPU liegen bzw. sofort anschliessend von der CPU verarbeitet werden.

Der externe Bus ist mit (0710) bezeichnet und es werden darüber zB aus einer damit verbundenen Datenquelle (z.B. Speicher, Peripherie) gelesen, bzw. an den externen Bus und der damit verbundenen Datensenke (z.B. Speicher, Peripherie) geschrieben. Dieser Bus kann insbesondere derselbe wie der externe Bus der CPU sein (0712 & gestrichelt).

Ein Protokoll (0711) zwischen Cache und Bus ist implementiert, das für einen korrekten Inhalt des Caches sorgt. Mit (0713)

ist ein FPGA (0713) bezeichent, der mit der VPU gekoppelt sein kann, um feingranulare Datenverarbeitung zu ermöglichen und/oder ein flexible adaptierbare Interface (0714) (z.B. diverse serielle Schnittstellen (V24, USB, etc.), diverse parallele Schnittstellen, Festplattenschnittstellen, Ethernet, Telekommunikationsschnittstellen (a/b, T0, ISDN, DSL, etc)) zu weiteren Baugruppen und/oder dem externen Bussystem (0712) zu ermöglichen.

Entsprechend Figur 8 befindet sich Speicherbereich des Betriebssystems eine Tabelle oder verkettete Liste (LINKLIST, 0801), die auf sämtliche VPUCALL-Tabellen (0802) in der Reihenfolge ihrer Erstellung zeigt.

Patentansprüche

1. Verfahren zur Übersetzung von Programmen auf ein System bestehend aus wenigstens einem ersten Prozessor und einer rekonfigurierbaren Einheit, dadurch gekennzeichnet, daß die Codeteile, die für die rekonfigurierbare Einheit geeignet sind, bestimmt und extrahiert und/oder separiert wird, wobei verbleibender Code zur Abarbeitung durch den ersten Prozessor bestimmt wird.

2. Verfahren nach Anspruch 1, dadurch gekennzeichnet, daß dem für den Prozessor extrahierten Code Interface-Code zugefügt wird, der eine Kommunikation zwischen Prozessor und rekonfigurierbarer Einheit entsprechend des Systemes ermöglicht.

3. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß dem für die rekonfigurierbarer Einheit extrahierten Code solcher Interface-Code zugefügt wird, der eine Kommunikation zwischen Prozessor und rekonfigurierbarer Einheit entsprechend des Systems ermöglicht.

4. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß der zu extrahierende Code aufgrund von automatisierten Analysen festgelegt wird.

5. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß Hinweise im Code zur Feststellung des zu extrahierenden Code automatisch ausgewertet werden.

6. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß der zu extrahierende Code aufgrund von Aufrufen von Unterprogrammen festgestellt wird.

7. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein Interface-Code vorgesehen wird, der eine Speicherkopplung (Shared-Memory) vorsieht und/oder eine Registerkopplung und/oder eine Kopplung mittels eines Netzwerkes bewirkt.

8. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß der extrahierte Code und/oder mit einer gegebenen Extraktion erzielbaren Resultate analysiert wird und gegebenenfalls die Extraktion mit neuen verbesserten Parametern erneut gestartet wird.

9. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß dem extrahierten Code Steuer-Code zur

Verwaltung und/oder Steuerung und/oder Kommunikation der Entwicklungssysteme zugefügt wird.

10. Verfahren nach einem der vorhergehenden Ansprüche, worin der erste Prozessor eine konventionelle Prozessorarchitektur aufweist, insbesondere ein Prozessor mit von - Neumann - und/oder Harwardarchitektur, Kontroller, CISC-, RISC-, VLIW-, DSP-Prozessor.

11. Verfahren insbesondere nach einem der vorhergehenden Ansprüche zur Übersetzung von Programmen auf ein System bestehend aus einem Prozessor und einer rekonfigurierbaren Einheit, dadurch gekennzeichnet, daß die Codeteile, die für die rekonfigurierbare Einheit geeignet sind, extrahiert werden, der verbleibende Code derart extrahiert wird, daß er mittels eines beliebigen gewöhnlichen unmodifizierten für den Prozessor geeigneten Compilers übersetzbar ist.

12. Vorrichtung zur Datenverarbeitung mit wenigstens einem herkömmlichen Prozessor und wenigstens einer rekonfigurierbaren Einheit, dadurch gekennzeichnet, daß ein Mittel zum Informationsaustausch, insbesondere von Daten- und Statusinformation, zwischen herkömmlichem Prozessor und rekonfigurierbarer Einheit aufweist, wobei das Mittel so ausgebildet ist, daß ein Daten- und Statusinformation zwischen denselben während der Abarbeitung eines oder mehrere Programme möglich ist und/oder ohne daß insbesondere die Datenverarbeitung auf dem rekonfigurierbaren Prozessor und/oder dem herkömmlichen Prozessor signifikant unterbrochen werden muß.

Fig. 1

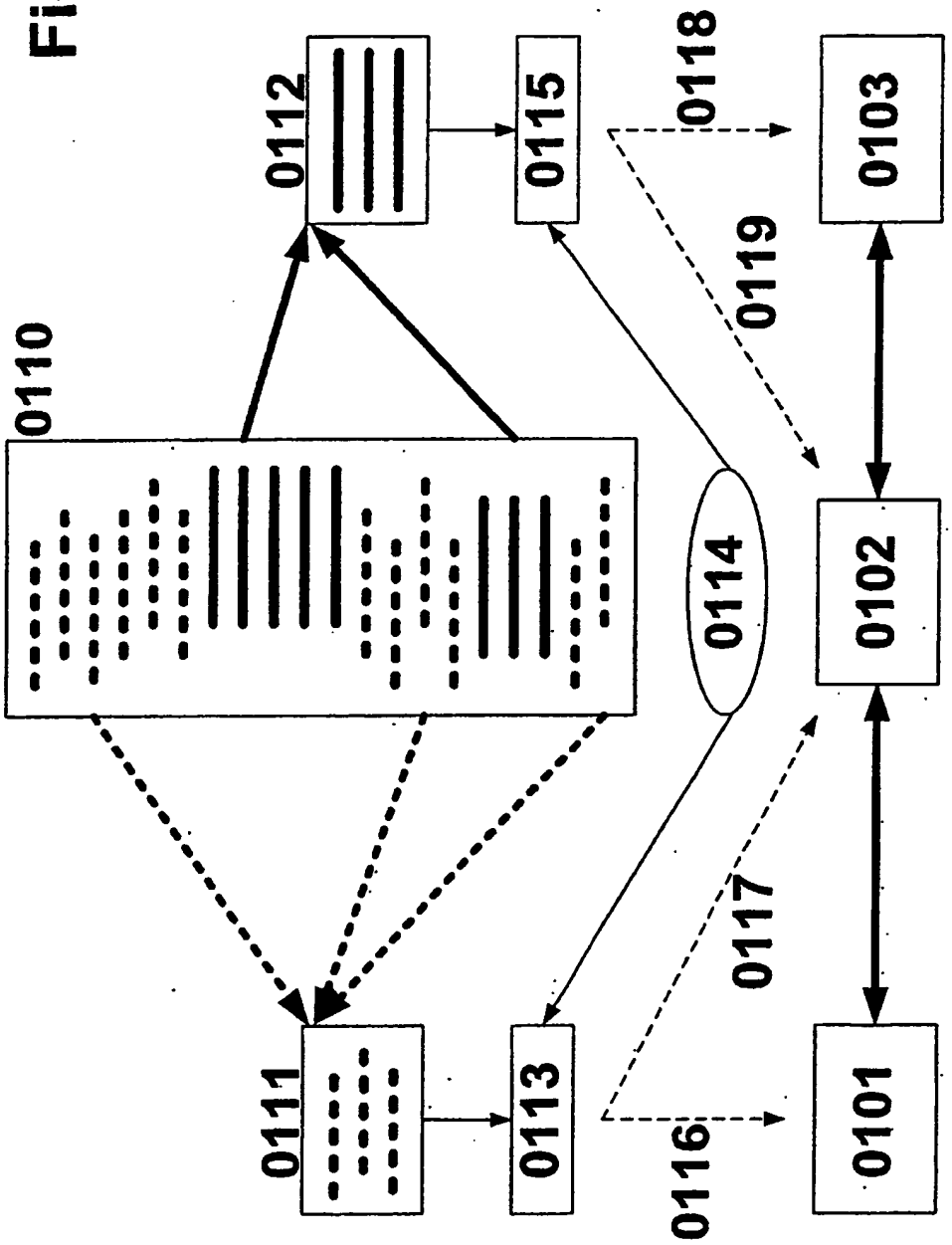
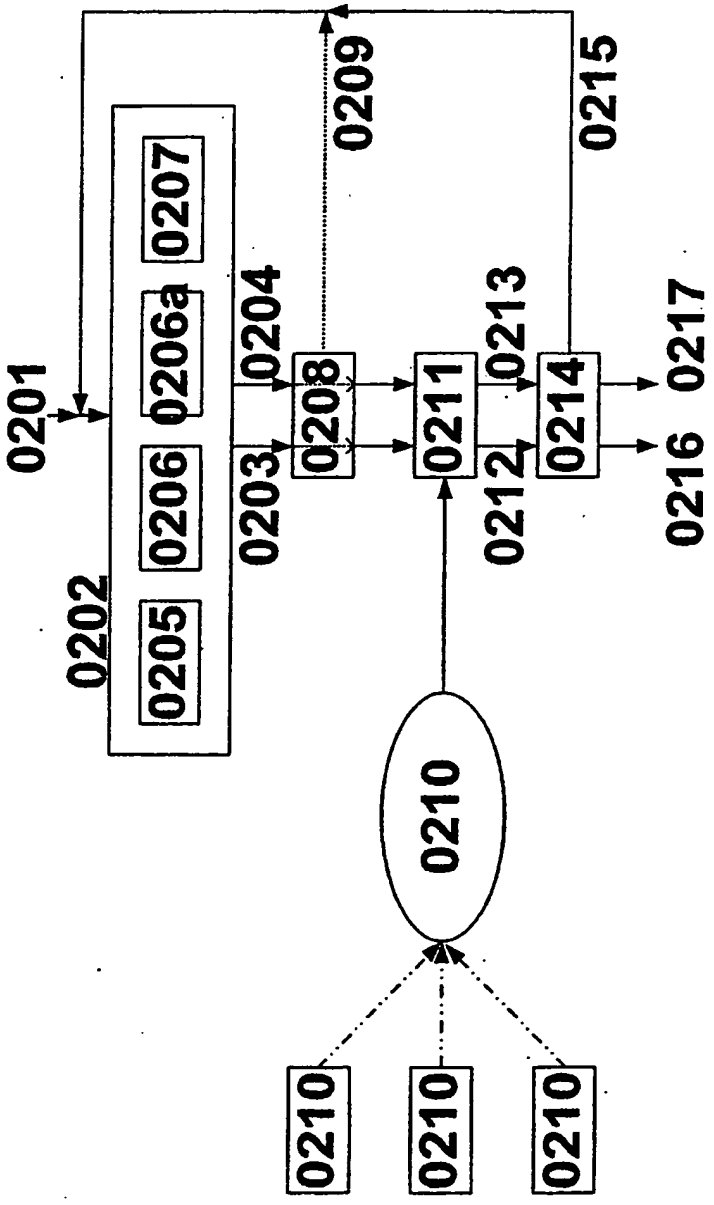

0110
0112
0115
0118
0103
0119
0114
0102
0111
0113
0117
0116
0101

Fig. 2


0201
0202
0203
0204
0205
0206
0206a
0207
0208
0209
0210
0210
0210
0211
0212
0213
0214
0215
0216
0217

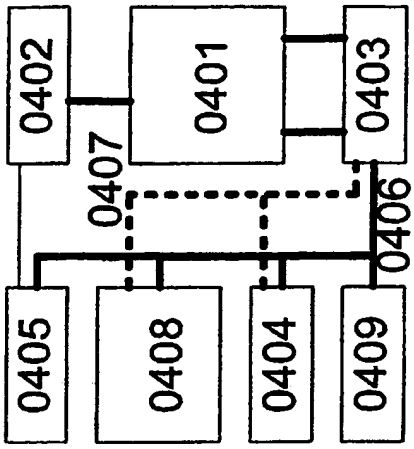

0402
0401
0403
0407
0406
0405
0408
0404
0409

Fig. 4

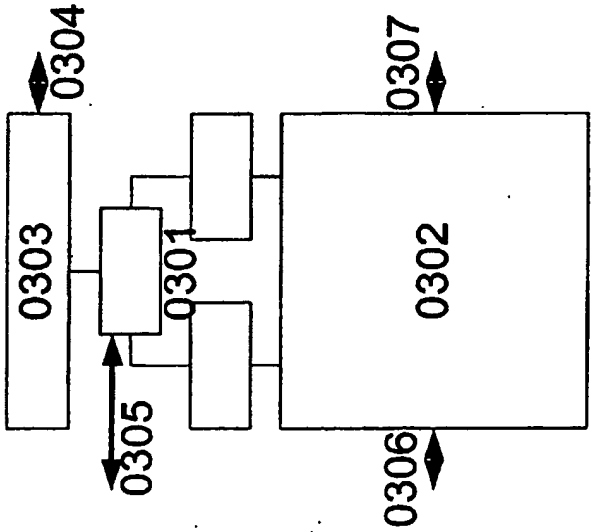

0304
0307
0303
0301
0302
0305
0306

Fig. 3

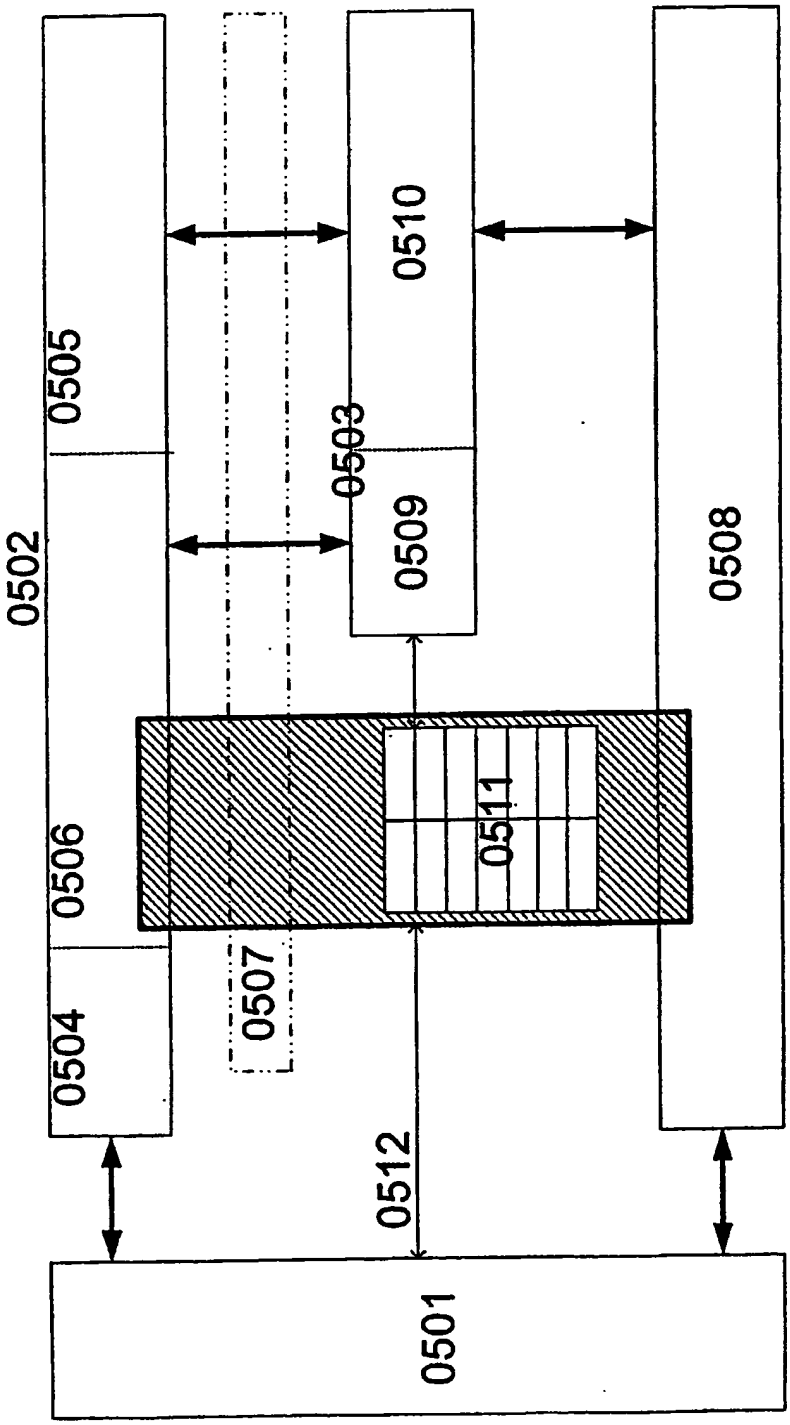


Fig. 5

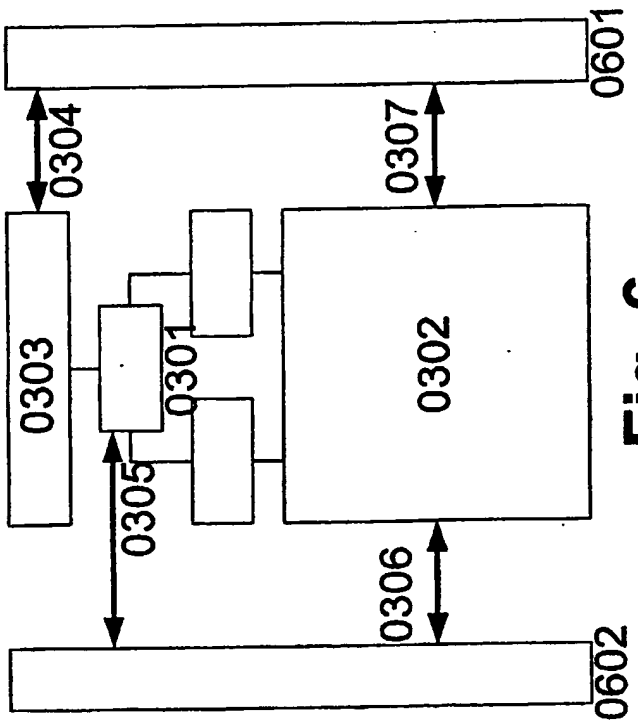


Fig. 6

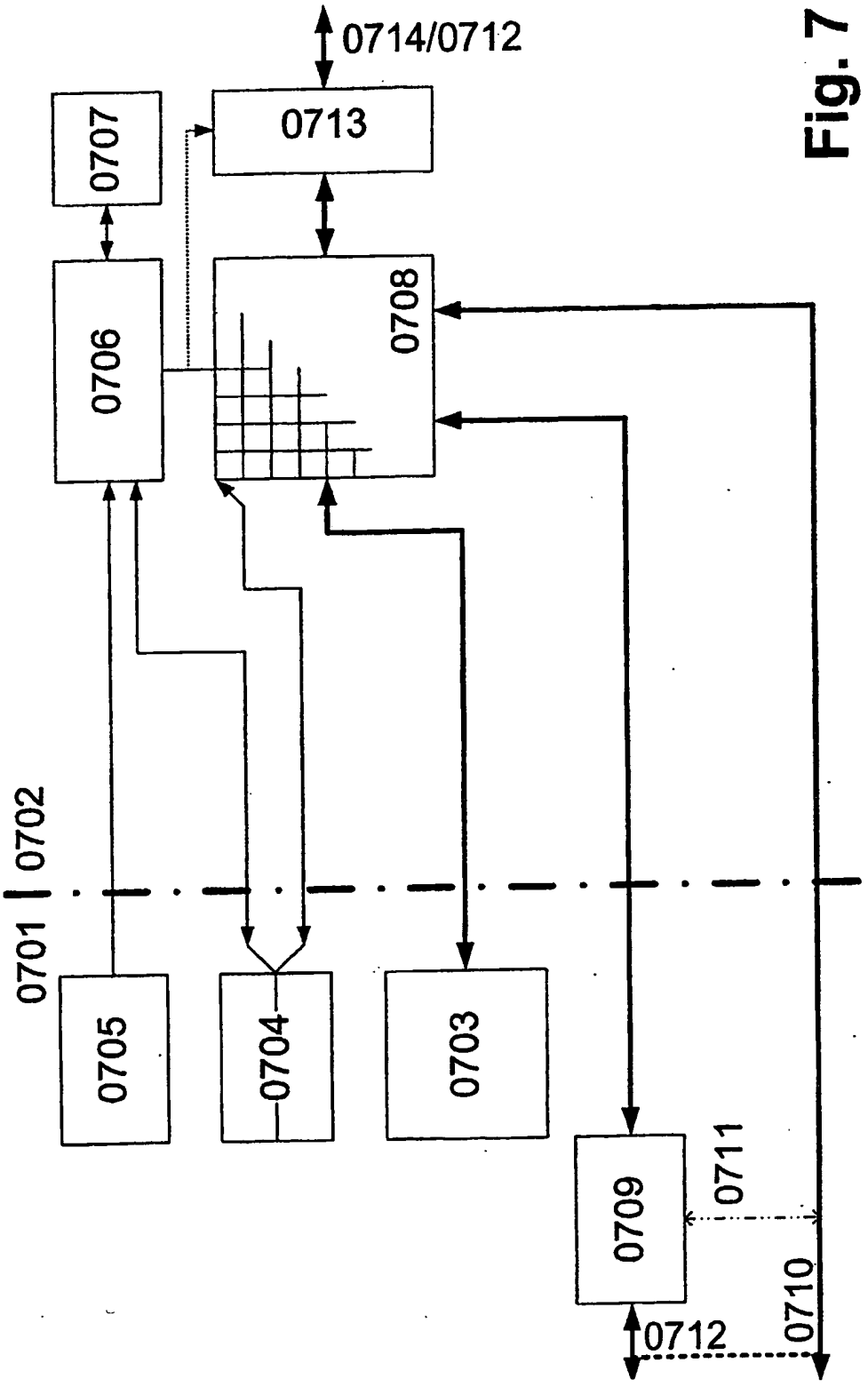

0714/0712
0713
0707
0706
0708
0702
0701
0705
0704
0703
0709
0711
0710
0712

Fig. 7

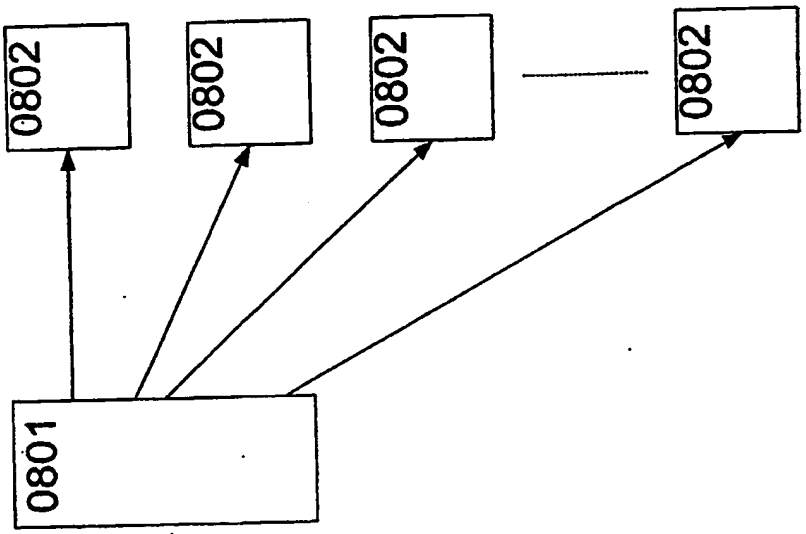
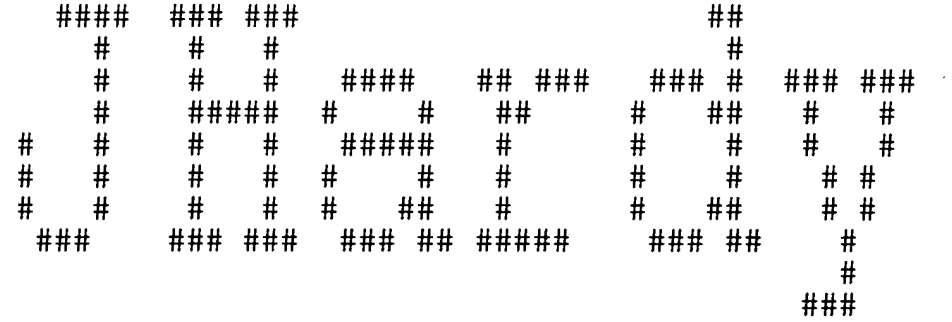
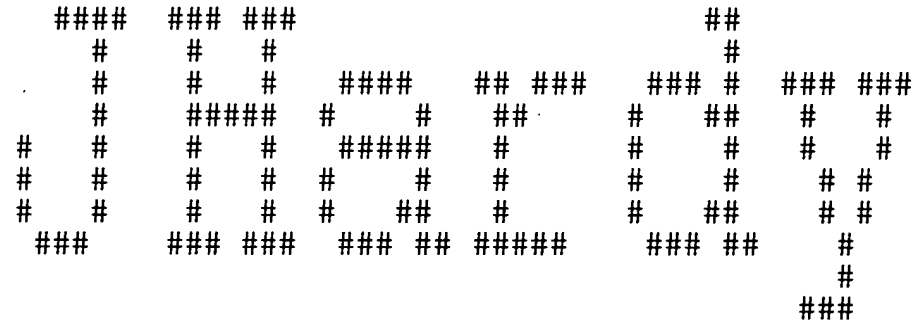


Fig. 8

****
****





****
****

Print Job Information:

Date: 11/13/2006

Time: 7:29:58 PM

Job Number: 621

(12) NACH DEM VERTRAG ÜBER DIE INTERNATIONALE ZUSAMMENARBEIT AUF DEM GEBIET DES PATENTWESENS (PCT) VERÖFFENTLICHTE INTERNATIONALE ANMELDUNG

(19) Weltorganisation für geistiges Eigentum
Internationales Büro

(43) Internationales Veröffentlichungsdatum
27. Februar 2003 (27.02.2003)

PCT

(10) Internationale Veröffentlichungsnummer
WO 03/017095 A2

(51) Internationale Patentklassifikation[7]: G06F 9/445

(21) Internationales Aktenzeichen: PCT/EP02/10065

(22) Internationales Anmeldedatum:
16. August 2002 (16.08.2002)

(25) Einreichungssprache: Deutsch

(26) Veröffentlichungssprache: Deutsch

(30) Angaben zur Priorität:

101 39 170.6	16. August 2001 (16.08.2001)	DE
101 42 903.7	3. September 2001 (03.09.2001)	DE
101 44 732.9	11. September 2001 (11.09.2001)	DE
101 45 792.8	17. September 2001 (17.09.2001)	DE
09/967,847	28. September 2001 (28.09.2001)	US
101 54 260.7	5. November 2001 (05.11.2001)	DE
102 07 225.6	21. Februar 2002 (21.02.2002)	DE
PCT/EP02/02398	5. März 2002 (05.03.2002)	EP
PCT/EP02/09131	15. August 2002 (15.08.2002)	EP

(71) Anmelder *(für alle Bestimmungsstaaten mit Ausnahme von US)*: PACT XPP TECHNOLOGIES AG [DE/DE]; Muthmannstrasse 1, 80939 München (DE).

(72) Erfinder; und

(75) Erfinder/Anmelder *(nur für US)*: VORBACH, Martin [DE/DE]; Gotthardstrasse 117a, 80689 München (DE). MAY, Frank [DE/DE]; An der Tuchbleiche 12, 81927 München (DE). NÜCKEL, Armin [DE/DE]; Drosselweg 4, 76777 Neupotz (DE).

(74) Anwalt: PIETRUK, Claus, Peter; Heinrich-Lilienfein-Weg 5, 76229 Karlsruhe (DE).

(81) Bestimmungsstaaten *(national)*: AE, AG, AL, AM, AT (Gebrauchsmuster), AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ (Gebrauchsmuster), CZ, DE (Gebrauchsmuster), DE, DK (Gebrauchsmuster), DK, DM, DZ, EC, EE (Gebrauchsmuster), EE, ES, FI (Gebrauchsmuster), FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK (Gebrauchsmuster), SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Bestimmungsstaaten *(regional)*: ARIPO-Patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), eurasisches Patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), europäisches Patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK, TR), OAPI-Patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Erklärung gemäß Regel 4.17:

— *hinsichtlich der Berechtigung des Anmelders, ein Patent zu beantragen und zu erhalten (Regel 4.17 Ziffer ii) für die folgenden Bestimmungsstaaten AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VC, VN, YU, ZA, ZM, ZW, ARIPO-Patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), eurasisches Patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), europäisches Patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK, TR), OAPI-Patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG)*

Veröffentlicht:

— *ohne internationalen Recherchenbericht und erneut zu veröffentlichen nach Erhalt des Berichts*

Zur Erklärung der Zweibuchstaben-Codes und der anderen Abkürzungen wird auf die Erklärungen ("Guidance Notes on Codes and Abbreviations") am Anfang jeder regulären Ausgabe der PCT-Gazette verwiesen.

(54) Title: METHOD FOR THE TRANSLATION OF PROGRAMS FOR RECONFIGURABLE ARCHITECTURES

(54) Bezeichnung: VERFAHREN ZUM ÜBERSETZEN VON PROGRAMMEN FÜR REKONFIGURIERBARE ARCHITEKTUREN

(57) Abstract: The invention relates to data processing with multidimensional fields and high-level language codes which can be used advantageously therefor .

(57) Zusammenfassung: Die Erfindung betrifft die Datenverarbeitung mit multidimensionalen Feldern und gibt an, wie hierfür vorteilhaft Hochsprachencodes einsetzbar sind.

WO 03/017095 A2

Titel: Verfahren zum Übersetzen von Programmen für rekonfigurierbare Architekturen

1. Einleitung

Die vorliegende Erfindung betrifft das Oberbegrifflich Beanspruchte. Damit befaßt sich die vorliegende Erfindung mit der Frage, wie rekonfigurierbare Architekturen optimal verwendet werden können und insbesondere damit, wie Anweisungen in einer gegebenen Hochsprache in rekonfigurierbaren Architekturen optimal zur Ausführung gebracht werden können.

Um in sog. Hochsprachen geschriebene Anweisungen zur Handhabung von Daten (Programme) in einer jeweiligen, zur Datenhandhabung verwendeten Architektur zur Ausführung zu bringen, sind sog. Compiler bekannt, die die Anweisungen der Hochsprache in an die verwendete Architektur besser angepaßte Anweisungen übersetzen. Compiler, die dabei hochparallele Architekturen besonders unterstützen, sind demgemäß parallelisierende Compiler.

Parallelisierende Compiler nach dem Stand der Technik verwenden für gewöhnlich spezielle Konstrukte wie Semaphore und/oder andere Verfahren zur Synchronisation. Dabei werden typischerweise technologiespezifische Verfahren verwendet. Bekannte Verfahren sind nicht geeignet, um funktional spezifizierte Architekturen mit dem zugehörigen Zeitverhalten und imperativ spezifizierte Algorithem zu kombinieren. Daher liefern die verwendeten Methoden nur in Spezialfällen zufriedenstellende Lösungen.

Compiler für rekonfigurierbare Architekturen, insbesondere für rekonfigurierbaren Prozessoren, verwenden für gewöhnlich Makros, die speziell für die bestimmte rekonfigurierbare Hardware erstellt wurden, wobei für die Erstellung der Makros zumeist Hardwarebeschreibungssprachen wie z.B. Verilog, VHDL oder System-C verwendet werden. Diese Makros werden dann von einer gewöhnlichen Hochsprache (z.B. C, C++) aus dem Programmfluss heraus aufgerufen (instantiiert).

Compiler für Parallelrechner sind bekannt, die auf einer grobgranularen Struktur, zumeist basierend auf kompletten Funktionen oder Threads Programmteile auf mehrere Prozessoren abbilden.

Weiterhin sind vektorisierende Compiler bekannt, die eine weitgehende lineare Datenverarbeitung, wie z.B. Berechnungen großer Ausdrücke in eine vektorisierte Form umwandeln und damit die Berechnung auf superskalaren Prozessoren und Vektorprozessoren (z.B. Pentium, Cray) ermöglichen.

Vorliegend wird ein Verfahren zur automatischen Abbildung von funktional oder imperativ formulierten Rechenvorschriften auf unterschiedliche Zieltechnologien beschrieben, insbesondere auf ASICs, rekonfigurierbare Bausteine (FPGAs, DPGAs, VPUs, ChessArray, KressArray, Chameleon, etc.; im folgenden unter dem Begriff VPU zusammengefaßt), sequentielle Prozessoren (CISC-/RISC-CPUs, DSPs, etc.; im folgenden unter dem Begriff CPU zusammengefaßt) und parallele Rechnersysteme (SMP, MMP, etc.). Hingewiesen wird insbesondere in diesem Zusammenhang auf die folgenden Schutzrechte und Patentanmeldungen desselben Anmelders: P 44 16 881.0-53, DE 197 81 412.3, DE 197 81 483.2, DE 196 54 846.2-53, DE 196 54 593.5-53, DE 197 04 044.6-53, DE 198 80 129.7, DE 198 61 088.2-53, DE 199 80 312.9, PCT/DE 00/01869, DE 100 36 627.9-33, DE 100 28 397.7,

DE 101 10 530.4, DE 101 11 014.6, PCT/EP 00/10516, EP 01 102 674.7, PACT13, PACT17, PACT18, PACT22, PACT24, PACT25, PACT26US, PACT02, PACT04, PACT08, PACT10, PACT15, PACT18(a), PACT 27, PACT19. Diese sind hiermit zu Offenbarungszwecken vollumfänglich eingegliedert.

VPUs bestehen grundsätzlich aus einer mehrdimensionalen homogenen oder inhomogenen, flachen oder hierarchischen Anordung (PA) von Zellen (PAEs), die beliebige Funktionen, i.b. logische und/oder arithmetische Funktionen und/oder Speicherfunktionen und/oder Netzwerkfunktionen ausführen können. Den PAEs ist typisch eine Ladeeinheit (CT) zugeordnet, die die Funktion der PAEs durch Konfiguration und ggf. Rekonfiguration bestimmt.

Das Verfahren basiert auf einem abstrakten parallelen Maschinenmodell, das neben dem endlichen Automaten auch imperative Problemspezifikationen integriert und eine effiziente algorithmische Ableitung einer Implementierung auf unterschiedliche Technologien ermöglicht.

Folgende Compilerklassen sind nach dem Stand der Technik bekannt:

Klassische Compiler, die häufig Stack-Maschinen-Code generieren und für sehr einfache Prozessoren geeignet waren, die im Wesentlichen als normale Sequenzer ausgestaltet sind. (vgl. N.Wirth, Compilerbau, Teubner Verlag).

Vektorisierender Compiler bauen weitgehend linearen Code, der auf spezielle Vektorrechner oder stark gepipelinte Prozessoren abgestimmt ist. Ursprünglich waren diese Compiler für Vektorrechner wie CRAY verfügbar. Moderne Prozessoren wie Pentium benötigen aufgrund der langen Pipelinestruktur ähnliche Verfahren. Da die einzelnen Rechenschritte vektorisiert (gepipe-

lined) ablaufen ist der Code sehr viel effizienter. Allerdings bereitet der bedingte Sprung Probleme für die Pipeline. Daher ist eine Sprungvorhersage sinnvoll, die ein Sprungziel annimmt. Ist die Annahme falsch, muss jedoch die gesamte Verarbeitungspipeline gelöscht werden. Mit anderen Worten ist jeder Sprung für diese Compiler problematisch, eine Parallelverarbeitung im eigentlichen Sinn ist nicht gegeben. Sprungvorhersagen und ähnliche Mechanismen erfordern einen erheblichen Zusatzaufwand an Hardware.

Grobgranulare parallele Compiler existieren im eigentlichen Sinne kaum, die Parallelität wird typischerweise durch den Programmierer oder das Betriebssystem markiert und verwaltet, also beipielsweise bei MMP-Computersysteme wie verscheiden IBM Architekturen, ASCI Red, etc. zumeist auf Thread-Ebene durchgeführt. Ein Thread ist ein weitgehend unabhängiger Programmblock oder gar ein anderes Programm. Threads sind daher grobgranular einfach zu parallelisieren. Die Synchronisation und Datenkonsistenz ist vom Programmierer bzw. dem Betriebssystem sicherzustellen. Dies ist aufwendig zu programmieren und erfordert einen wesentlichen Anteil der Rechenleistung eines Parallelrechner. Zudem ist durch diese grobe Parallelisierung nur ein Bruchteil der eigentlich möglichen Parallelität tatsächtlich nutzbar.

Feingranulare parallele (z.B. VLIW) Compiler versuchen die Parallelität feingraunlar in VLIW Rechenwerke abzubilden, die mehrere Rechenoperationen in einem Takt parallel ausführen können aber einen gemeinsamen Registersatz besitzen. Ein wesentliches Problem stellt dieser limitierte Registersatz dar, da er die Daten für sämtliche Rechenoperationen bereitstellen muss. Zudem erschweren Datenabhängigkeiten und inkonsistente Lese/Schreiboperationen (LOAD/STORE) die Parallelisierung.

Rekonfigurierbare Prozessoren weisen eine große Anzahl an unabhängigen Rechenwerken auf, die typisch in einem Feld angeordnet sind. Diese sind typisch nicht durch einen gemeinsamen Registersatz, sondern durch Busse miteinander verbunden. Dadurch lassen sich einerseits leicht Vektorrechenwerke aufbauen, andererseits können auch einfach parallele Operationen durchgeführt werden. Durch die Busverbindungen werden entgegen der herkömmlichen Registerkonzepte Datenabhängigkeiten aufgelöst.

Die Aufgabe der vorliegenden Erfindung besteht darin, Neues für die gewerbliche Anwendung bereitzustellen.

Die Lösung dieser Aufgabe wird in unabhängiger Form beansprucht. Bevorzugte Ausführungsformen finden sich in den Unteransprüchen.

Es wird also vorgeschlagen, für einen Compiler für rekonfigurierbare Prozessoren die Konzepte von vektorisierenden Compilern und parallelisierenden (z.B. VLIW) Compilern zugleich anzuwenden und somit auf feingranularer Ebene zu Vektorisieren und parallelisieren.

Ein wesentlicher Vorteil besteht darin, dass der Compiler nicht auf eine fest vorgegebene Hardwarestruktur abbilden muss, sondern die Hardwarestruktur mit dem erfindungsgemäßen Verfahren so konfiguriert werden kann, dass sie optimal für die Abbildung des jeweiligen compilierten Algorithmus geeignet ist.

2. Beschreibung

Als Grundlage zur Abarbeitung praktisch jeder Methodik zur Spezifizierung von Algorithmen wird der endliche Automat genutzt. Die Struktur eines endlichen Automaten ist in Figur 1

abgebildet. Ein einfacher endlicher Automat zerfällt in ein kombinatorisches Netz und eine Registerstufe zum Zwischenspeichern von Daten zwischen den einzelnen Datenverarbeitungszyklen.

Ein endlicher Automat führt eine Anzahl rein kombinatorischer (also z.B. logischer und/oder arithmetischer) Datenmanipulationen aus, um danach einen stabilen Zustand zu erreichen, der in einem Register(satz) repräsentiert wird. Basierend auf diesem stabilen Zustand wird entschieden, welcher nächste Zustand im nächsten Verarbeitungsschritt erreicht werden soll und somit auch, welche kombinatorischen Datenmanipulationen im nächsten Schritt durchgeführt werden sollen.

Beispielsweise repräsentiert ein Prozessor oder Sequenzer einen endlichen Automaten. In einem ersten Verarbeitungsschritt kann eine Subtraktion von datendurchgeführt werden. Das Ergebnis wird gespeichert. Im nächsten Schritt kann basierend auf dem Ergebnis der Subtraktion ein Sprung durchgeführt werden, der je nach Vorzeichen des Ergebnisses in eine andere Weiterverarbeitung führt.

Der endliche Automat ermöglicht die Abbildung komplexer Algorithmen auf beliebige sequentielle Maschinen, wie in Figur 2 abgebildet. Der dargestellte komplexe endliche Automat besteht aus einem komplexen kombinatorischen Netz, einem Speicher zum Ablegen von Daten und einem Adressgenerator zum Adressieren der Daten im Speicher.

Nun kann jedes beliebige sequentielle Programm grundlegend als endlicher Automat interpretiert werden, wobei aber zumeist ein sehr großes kombinatorisches Netz entsteht. Bei der Programmierung klassischer "von Neumann"-Architekturen - also bei allen CPUs - werden daher die kombinatorischen Operationen in eine Folge von jeweils einzelnen einfachen, fest vorgegebenen

Operationen (OpCodes) auf CPU-interne Register zerlegt. Durch diese Zerlegung entstehen Zustände zur Steuerung der in eine Folge zerlegten kombinatorischen Operation, die aber innerhalb der ursprünglichen kombinatorischen Operation per se nicht vorhanden sind, bzw. nicht benötigt werden. Daher sind jedoch die abzuarbeitenden Zustände einer von Neumann Maschinen grundsätzlich von den algorithmischen Zuständen eines kombinatorischen Netzes, also den Registern endlicher Automaten zu unterscheiden.

Es wurde nun erkannt, dass die VPU-Technologie (wie sie im Wesentlichen durch einige oder alle der Schriften PACT01, PACT02, PACT03, PACT04, PACT05, PACT08, PACT10, PACT13, PACT17, PACT18, PACT22, PACT24 definiert ist, die durch Bezugnahme vollumfänglich eingegliedert sind) im Gegensatz zu den starren OpCodes von CPUs ermöglicht, komplexe Instruktionen entsprechend eines abzubildenden Algorithmus wie in flexiblen Konfigurationen hineinzukompilieren.

2.1 Arbeitsweise des Compilers

Besonders vorteilhaft ist bei der Arbeitsweise des Compilers, wenn die komplexen Instruktionen derart generiert werden, daß diese möglichst lange in der PAE-Matrix ohne Rekonfiguration ausgeführt wird.

Der Compiler generiert weiterhin den endlichen Automaten bevorzugt aus dem imperativen Quelltext derart, daß er der jeweiligen PAE-Matrix besonders gut angepasst ist, also solche Operationen darin vorgesehen werden, die die typisch grobgranularen Logikkreise (ALUs etc.), gegebenenfalls auch vorhandene feingranulare Elemente (FPGA-Zellen in der VPU, statemachines etc.) besonders effizient nutzen.

Der compilererzeugte endliche Automat wird dann in Konfigurationen zerlegt.

Das Abarbeiten (Interpretieren) des endlichen Automaten geschieht auf einer VPU derart, daß die generierten Konfigurationen succesive auf die PAE-Matrix abgebildet werden und die Arbeitsdaten und/oder Zustände, die zwischen den Konfigurationen zu übertragen sind, in Speicher abgelegt werden. Dazu kann das aus PACT04 bekannte Verfahren bzw. die entsprechende Architektur verwendet werden. Dieser Speicher wird vom Compiler bestimmt beziehungsweise vorgesehen. Es repräsentiert eine Konfiguration dabei eine Mehrzahl von Instruktionen; eine Konfiguration bestimmt zugleich für eine Vielzahl von Takten die Arbeitsweise der PAE-Matrix, während dieser Takte wird eine Vielzahl von Daten in der Matrix verarbeitet; diese stammen aus einer VPU externen Quelle und/oder einem internen Speicher und werden an eine externe Quelle und/oder einen internen Speicher geschrieben. Die internen Speicher ersetzen dabei den Registersatz einer CPU nach dem Stand der Technik derart, daß z.B. ein Register durch einen Speicher repräsentiert wird, wobei nicht ein Datenwort je Register gespeichert wird, sondern ein gesamter Datensatz je Speicher.

Wesentlich kann auch sein, daß die Daten und/oder Zustände der Verarbeitung einer ablaufenden Konfiguration compilerbestimmt in die Speicher abgelegt werden und somit der nächsten ablaufenden Konfiguration zur Verfügung stehen.

Ein bedeutender Unterschied zu Compilern, die auf Instruktionsbasis parallelisieren, besteht somit darin, daß das Verfahren die PAE-Matrix derart konfiguriert und rekonfiguriert, dass eine konfigurierte Folge von kombinatorischen Netzen auf einer VPU emuliert wird, während herkömmliche Compiler geladene Folgen von Instruktionen (OpCodes) kombinieren, wobei eine

Instruktion als ein kombinatorisches Netz betrachtet werden kann.

2.2 Ausführungsbeispiel WHILE-Sprache

Im Folgenden soll die Funktionsweise des Compilers anhand einer einfachen Sprache beispielhaft verdeutlicht werden. Dabei wird von einer Sprache ausgegangen, die in ihren Grundlagen bereits bekannt ist, wobei in einer bekannten Veröffentlichung [Referenz "Doktorarbeit Armin Nückel"] jedoch lediglich die Abbildung einer Funktion auf ein statisches kombinatorisches Netz beschrieben wird, während mit der Erfindung nun die Abbildung auf Konfigurationen erfolgt, die dann in einer zeitlichen Reihenfolge entsprechend des Algorithmus und der sich während der Verarbeitung ergebenden Zustände auf die PAE-Matrix abgebildet werden.

Die Programmiersprache geht davon aus, dass neben einfachen logischen und/oder arithmetischen Verknüpfungen ein Befehl "WHILE" existiert, der mit folgender Syntax definiert ist:

WHILE...

Mögliche Konstrukte sind damit: Anweisung
Folge von Anweisungen
Schleife

Eine Anweisung oder eine Folge von Anweisungen ist durch das beschriebene Compiler-Verfahren auf ein kombinatorisches Netz abbildbar.

Figur 3a zeigt ein kombinatorisches Netz mit den dazugehörenden Variablen. Dabei kann sich der Inhalt ein und derselben Variable (z.B. x1) von einer Stufe (0301) des Netzes zur nächsten (0302) ändern.

Diese Veränderung ist beispielsweise für die Zuweisung

x1 := x1 + 1

in **Figur 3b** dargestellt.

Zur Adressierung zum Lesen der Operanden und zum Speichern der Ergebnisse können nun Adressgeneratoren mit dem kombinatorischen Netz der Zuweisung synchronisiert werden. Mit jeder verarbeiteten Variable werden entsprechende neue Adressen für Operanden und Ergebnisse generiert (**Figur 3c**). Die Art des Adressgenerators ist prinzipiell beliebig und hängt von den Adressierungsschematas der compilierten Applikation ab.
Für Operanden und Ergebnisse können gemeinsame, kombinierte oder vollständig unabhängige Adressgeneratoren implementiert werden.

Typischerweise werden bei der Datenverarbeitung wie im vorliegenden Datenverarbeitungsmodell eine Mehrzahl von Daten innerhalb einer bestimmten Konfiguration der PAEs verarbeitet. Bevorzugt ist der Compiler daher für die in vielen, wenn nicht den meisten Anwendungen möglichen einfachen FIFO-Modus ausgelegt, der zumindest für die Datenspeicher anwendbar ist, die innerhalb dieser Beschreibung zum Speichern von Daten und Zuständen der Datenverarbeitung (quasi als Ersatz eines gewöhnlichen Registersatzes herkömmlicher CPUs) dienen. Mit anderen Worten dienen Speicher der temporären Speicherung von Variablen zwischen den Konfigurationen. Auch hier ist eine Konfiguration ähnlich einer Instruktion eines normalen Prozessors und die Speicher (insbesondere eine Mehrzahl) sind vergleichbar mit dem Registersatz eines normalen Prozessors.

2.2.3 Folgen von Anweisungen

Eine Folge der beispielhaften Zuweisung läßt sich wie folgt generieren (**Figur 4a**):

```
x1 := 0;
WHILE TRUE DO
    x1 := x1 + 1;
```

Diese Folge läßt sich nunmehr mittels einer Zuweisung gemäß 2.2.1 und Adressgeneratoren für Operanden und Ergebnisse abbilden.

Endliche Folgen

Der Vollständigkeit halber soll eine besondere Ausgestaltung von Folgen abseits der definierten Konstrukte der WHILE Sprache diskutiert werden. Eine endliche Folge der beispielhaften Zuweisung läßt sich wie folgt generieren:

```
FOR i:=1 TO 10
    x1 := x1 + 1;
```

Eine derartige Folge läßt sich durch zwei Arten implementieren:

a) Durch Generierung eines Addierers zur Berechnung von i entsprechend des WHILE-Konstruktes (siehe 2.2.4) und eines weiteren Addierers zur Berechnung von x1. Die Folge wird als Schleife abgebildet und iterativ berechnet (Figur 5a).
b) Durch Auswalzen der Schleife, wodurch die Berechnung von i als Funktion entfällt. Die Berechnung von x1 wird i-mal instantiiert und als Pipeline aufgebaut, wodurch i nacheinander geschaltete Addierer entstehen (**Figur 5b**).

2.2.4 Bedingungen

Bedingungen lassen sich mittels WHILE ausdrücken. Beispielsweise:

```
x1 := 0;
WHILE x1 < 10 DO
    x1 := x1 + 1;
```

Die Abbildung generiert eine zusätzliche PAE zur Verarbeitung des Vergleiches. Das Vergleichsergebnis wird durch ein Statussignal repräsentiert (vgl. Trigger in PACT08), das von den

die Anweisung verarbeitenden PAEs und den Adressgeneratoren ausgewertet wird.
Die resultierende Abbildung ist in **Figur 4b** dargestellt.

Durch die Auswertung der Bedingung (hier WHILE, generell und einleuchtenderweise auch andere Anweisungen wie IF; CASE realisierbar) wird ein Status generiert, der der nachfolgenden Datenverarbeitung zur Verfügung gestellt werden kann (DE 197 04 728.9) und/oder an die CT oder eine lokale Ladesteuerung (DE 196 54 846.2) gesendet werden kann, die daraus Information über den weiteren Programmfluß und evtl. anstehende Rekonfigurationen ableitet.

2.2.5 Grundlegendes Verfahren

Entsprechend den vorherigen Verfahren wird jedes Programm in ein System abgebildet, das wie folgt aufgebaut ist:
1. Speicher für Operanden (vgl. Register einer CPU)
2. Speicher für Ergebnisse (vgl. Register einer CPU)
3. Netzwerk aus a) Zuweisungen und/oder b) Vergleichen-Anweisungen, also Bedingungen wie z.B. IF, CASE, Schleifen (WHILE, FOR, REPEAT)
4. Optionalen Adressgenerator(en) zur Ansteuerung der Speicher nach 1 und 2.

2.2.6 Umgang mit Zuständen

Es wird nun für die Zwecke des beschriebenen Compilers zwischen algorithmisch relevanten und irrelevanten Zuständen unterschieden. Zustände werden in der VPU-Technologie für gewöhnlich durch Statussignale (z.B. Trigger in PACT08) und/oder Handshakes (z.B. RDY/ACK in PACT02) dargestellt. Generell können Zustände (v.a. in anderen Technologien, wie FPGAs, DPGAs, Chameleon-Bausteinen, Morphics, etc.) durch beliebige Signale,

Signalbündel und/oder Register dargestellt werden. Das offenbarte Compilierverfahren kann auch auf solche angelegt werden, obwohl wesentliche Teile der Beschreibung bevorzugt auf die VPU fokussieren.
Relevante Zustände sind innerhalb des Algorithmus notwendig um dessen korrekte Funktion zu beschreiben. Sie sind für den Algorithmus wesentlich.
Irrelevante Zustände entstehen durch die verwendete Hardware und/oder die gewählte Abbildung oder aus anderen sekundären Gründen. Sie sind damit für die Abbildung (also die Hardware) wesentlich.

Lediglich die relevanten Zustände müssen mit den Daten erhalten werden. Daher werden diese zusammen mit den Daten in den Speichern abgelegt, da sie entweder als Ergebnis der Verarbeitung mit den Daten auftraten oder als Operanden mit den Daten für den nächsten Verarbeitungszyklus notwendig sind.

Irrelevante Zustände sind dagegen nur örtlich und/oder zeitlich lokal notwendig und müssen daher nicht gespeichert werden.

Beispiel:

a) Die Zustandsinformation eines Vergleichs ist für die weitere Verarbeitung der Daten relevant, da dieser die auszuführenden Funktionen bestimmt.
b) Angenommen, ein sequentieller Dividierer entsteht beispielsweise durch Abbildung eines Divisionsbefehles auf eine Hardware, die nur die sequentielle Division unterstützt. Dadurch entsteht ein Zustand, der den Rechenschritt innerhalb der Division kennzeichnet. Dieser Zustand ist irrelevant, da für den Algorithmus nur das Ergebnis (also die ausgeführte Division) erforderlich ist. In diesem Fall werden also lediglich das Ergebnis und die Zeitinformation (also die Verfügbarkeit) benötigt. Der Compiler unterschei-

det solche relevante und irrelevante Zustände bevorzugt voneinander.

Die Zeitinformation ist beispielsweise in der VPU-Technologie nach PACT01, 02, 13 durch das RDY/ACK Handshake erhältlich. Hierzu ist jedoch besonders anzumerken, dass das Handshake ebenfalls keine relevanten Zustand darstellt, da es lediglich die Gültigkeit der Daten signalisiert, wodurch sich wiederum die verbleibende relevante Information auf die Existenz gültiger Daten reduziert.

2.2.7 Umgang mit Zeit

In vielen Programmiersprachen, besonders in sequentiellen wie z.B. C, wird eine exakte zeitliche Reihenfolge implizit durch die Sprache vorgegeben; bei sequentiellen Programmiersprachen geschieht dies beispielsweise durch die Reihenfolge der einzelnen Anweisungen. Sofern dies durch die Programmiersprache und/oder den Algorithmus erforderlich ist, wird das Compilierverfahren so ausgeführt, dass sich die Zeitinformation auf Synchronisationsmodelle wie RDY/ACK und/oder REQ/ACK oder ein Time-Stamp-Verfahren nach DE 101 10 530.4 abbilden lässt.

Mit anderen Worten wird die implizite Zeitinformation von sequentiellen Sprachen in ein Handshake Protokoll derart abgebildet, dass das Handshake Protokoll (RDY/ACK-Protokoll) die Zeitinformation überträgt und insbesondere die Reihenfolge der Zuweisungen garantiert.

Beispielsweise wird die nachfolgende for-Schleife nur dann durchlaufen und iteriert, wenn die Variable inputstream je Durchlauf mit einem RDY quittiert ist. Bleibt RDY aus, wird der Schleifendurchlauf bis zum Eintreffen von RDY angehalten.
while TRUE

```
s := 0
for i:= 1 to 3
    s := s + inputstream;
```

Die Eigenschaft der sequentiellen Sprachen, nur von der Befehlsverarbeitung gesteuert zu werden, wird somit bei der Compilierung mit dem Datenflußprinzip, die Verarbeitung durch den Datenstrom, bzw. die Existenz von Daten zu steuern verbunden. Mit anderen Worten wird ein Befehl und/oder eine Anweisung (z.B. s := s + inputstream;) nur verarbeitet, wenn die Operation ausgeführt werden kann und die Daten verfügbar sind.

Bemerkenswert ist, daß dieses Verfahren gewöhnlicherweise zu keiner Änderung der Syntax oder Semantik einer Hochsprache führt. Es kann also vorhandener Hochsprachencode durch Neucompilierung problemfrei zur Ausführung auf einer VPU gebracht werden.

2.2.8 Laden und Speichern von Daten

Für die Grundlagen der Load/Store Operationen ist folgendes beachtlich.

Die nachfolgenden Adressierungsarten werden unterschiedlich behandelt:

1. externe Adressierung, also die Datentransfers mit externen Baugruppen
2. interne Adressierung, also die Datentransfers zwischen PAEs, i.b. zwischen RAM-PAEs und ALU-PAEs

Desweiteren kann die zeitliche Entkopplung der Datenverarbeitung und dem Laden und Speichern der Daten besondere Beachtung finden.

Bustransfers werden in interne und externe Transfers zerlegt.

bt1) Externe Lesezugriffe (Load Operation) werden separiert, in einer möglichen Ausführung auch bevorzugt in eine separate Konfiguration übersetzt. Die Daten werden von einem externen Speicher in einen internen transferiert.

bt2) Interne Zugriffe werden mit der Datenverarbeitung gekoppelt, d.h. die internen Speicher (Register Operation) werden für die Datenverarbeitung gelesen, bzw. beschrieben.

bt3) Externe Schreibzugriffe (Store Operation) werden separiert, in einer bevorzugten möglichen Ausführung auch in eine separate Konfiguration übersetzt. Die Daten werden von einem internen Speicher in einen externen transferiert.

Wesentlich ist, dass die bt1, bt2, bt3 - also das Laden der Daten (Load), das Verarbeiten der Daten (Datenverarbeitung und bt2) und das Schreiben der Daten (bt3) - in unterscheidliche Konfigurationen übersetzt werden können und diese ggf. zu einem unterschiedlichen Zeitpunkt ausgeführt werden.

Das Verfahren soll an dem nachfolgenden Beispiel verdeutlicht werden:

```
function example (a, b : integer) -> x : integer
for i:= 1 to 100
  for j:= 1 to 100
    x[i] := a[i] * b[j]
```

Die Funktion kann vom Compiler in drei Teile, bzw. Konfigurationen (subconfig) transformiert:
example#dload: Lädt die Daten von extern (Speicher, Peripherie, etc.) und schreibt diese in interne Speicher. Interne Speicher sind mit r# und dem Namen der ursprünglichen Variable gekennzeichnet.

example#process: Entspricht der eigentlichen Datenverarbeitung. Diese liest die Daten aus internem Operanden und schreibt die Ergebnisse wieder in interne Speicher.
example#dstore: schreibt die Ergebnisse aus dem internen Speicher nach extern (Speicher, Peripherie, etc.).

```
function example# (a, b : integer) -> x : integer
subconfig example#dload
for i := 1 to 100
  r#a[i] := a[i]
for j := 1 to 100
  r#b[j] := b[j]

subconfig example#process
for i := 1 to 100
  for j:= 1 to 100
    r#x[i] := r#a[i] * r#b[j]

subconfig example#dstore
for i := 1 to 100
  x[i] := r#x[i]
```

Ein wesentlicher Effekt des Verfahrens ist, dass anstatt i*j = 100 * 100 = 10.000 externe Zugriffe nur i+j = 100 + 100 = 200 externe Zugriffe zum Lesen der Operanden ausgeführt werden. Diese Zugriffe sind zudem noch vollkommen linear, was die Transfergeschwindigkeit bei modernen Bussystemen (Burst) und/oder Speichern (SDRAM, DDRAM, RAMBUS, etc) erheblich beschleunigt.

Die internen Speicherzugriffe erfolgen parallel, da den Operanden unterschiedliche Speicher zugewiesen wurden.

Zum Schreiben der Ergebnisse sind i = 100 externe Zugriffe notwendig, die ebenfalls wieder linear mit maximaler Performance erfolgen können.

Wenn die Anzahl der Datentransfers vorab nicht bekannt ist (z.B. WHILE-Schleifen) oder sehr groß ist, kann ein Verfahren verwendet werden, das bei Bedarf durch Unterprogrammaufrufe die Operanden nachlädt bzw. die Ergebnisse nach extern schreibt. Dazu können in einer bevorzugten Ausführung (auch) die Zustände der FIFOs abgefragt werden: 'empty' wenn das FIFO leer ist, sowie 'full' wenn das FIFO voll ist. Entsprechend der Zustände reagiert der Programmfluß. Zu bemerken ist, dass bestimmte Variablen (z.B. ai, bi, xi) global definiert sind. Zu Performanceoptimierung kann ein Scheduler entsprechend der bereits beschriebenen Verfahren die Konfigurationen example#dloada, example#dloadb bereits vor dem Aufruf von example#process bereits ausführen, sodass bereits Daten vorgeladen sind. Ebenso kann example#dstore(n) nach der Terminierung von example#process noch aufgerufen werden um r#x zu leeren.

```
subconfig example#dloada(n)
  while !full(r#a) AND ai<=n
    r#a[ai] := a[ai]
    ai++

subconfig example#dloadb(n)
  while !full(r#b) AND bi<=n
    r#b[bi] := b[bi]
    bi++

subconfig example#dstore(n)
  while !empty(r#x) AND xi<=n
    x[xi] := r#x[xi]
    xi++
```

```
subconfig example#process
for i := 1 to n
  for j:= 1 to m
    if empty(r#a) then example#dloada(n)
    if empty(r#b) then example#dloadb(m)
    if full(r#x) then example#dstore(n)

    r#x[i] := r#a[i] * r#b[j]
  bj := 1
```

Die Unterprogrammaufrufe und das Verwalten der globalen Variablen sind für rekonfigurierbare Architekturen vergleichsweise aufwendig. Daher kann in einer bevorzugten Ausführung die nachfolgende Optimierung durchgeführt werden, in welcher sämtliche Konfigurationen weitgehend unabängig ablaufen und nach vollständiger Abarbeitung terminieren (terminate). Da die Daten b[j] mehrfach erforderlich sind, muß example#dloadb entsprechend mehrfach durchlaufen werden. Dazu werden beispielsweise zwei Alternativen dargestellt:
Alternative 1: example#dloadb terminiert nach jedem Durchlauf und wird von example#process für jeden Neustart neu konfiguriert.
Alternative 2: example#dloadb läuft unendlich und wird von example#process terminiert.

Während 'idle' ist eine Konfiguration untätig (wartend).

```
subconfig example#dloada(n)
  for i:= 1 to n
    while full(r#a)
      idle
    r#a[i] := a[i]
  terminate

subconfig example#dloadb(n)
```

```
  while 1 // ALTERNATIVE 2
    for i:= 1 to n
      while full(r#b)
        idle
      r#b[i] := a[i]
  terminate

subconfig example#dstore(n)
  for i:= 1 to n
    while empty(r#b)
      idle
    x[i] := r#x[i]
  terminate

subconfig example#process
for i := 1 to n
  for j:= 1 to m
    while empty(r#a) or empty(r#b) or full(r#x)
      idle
    r#x[i] := r#a[i] * r#b[j]
  config example#dloadb(n)  // ALTERNATIVE 1
terminate example#dloadb(n)  // ALTERNATIVE 2
terminate
```

Zur Vermeidung von Wartezyklen können Konfigurationen auch terminiert werden, sobald sie ihre Aufgabe temporär nicht weiter erfüllen können. Die entsprechende Konfiguration wird von dem rekonfigurierbaren Baustein entfernt, verbleibt jedoch im Scheduler. Hierzu wird im Folgenden der Befehl 'reenter' verwendet. Die relevanten Variablen werden vor der Terminierung gesichert und bei der wiederholten Konfiguration wiederhergestellt:

```
subconfig example#dloada(n)
```

```
  for ai:= 1 to n
    if full(r#a) reenter
    r#a[ai] := a[ai]
  terminate

subconfig example#dloadb(n)
  while 1 // ALTERNATIVE 2
    for bi:= 1 to n
      if full(r#b) reenter
      r#b[bi] := a[bi]
  terminate

subconfig example#dstore(n)
  for xi:= 1 to n
    if empty(r#b) reenter
    x[xi] := r#x[xi]
  terminate

subconfig example#process
  for i := 1 to n
    for j:= 1 to m
      if empty(r#a) or empty(r#b) or full(r#x) reenter
      r#x[i] := r#a[i] * r#b[j]
    config example#dloadb(n)  // ALTERNATIVE 1
  terminate example#dloadb(n)  // ALTERNATIVE 2
  terminate
```

2.3 Makros

Komplexere Funktionen einer Hochsprache, wie z.B. Schleifen, werden typisch durch Makros realisiert. Die Makros werden dabei vom Compiler vorgegeben und zur Übersetzungszeit instantiiert. (vgl. Figur 4).

Die Makros sind entweder aus einfachen Sprachkonstrukten der Hochsprache oder auf Assemblerlevel aufgebaut. Makros können parametrisiert sein, um eine einfache Adaption an den beschiebenen Algorithmus zu ermöglichen. (vgl. Figur 5, 0502). Die Makros sind auch vorliegend einzugliedern.

2.4 Feedback Loops und Register

Innerhalb der Abbildung eines Algorithmus in ein kombinatorisches Netz können unverzögerte Rückkopplungen entstehen, die unkontrolliert schwingen.
In praktisch implementierten VPU-Technologien gemäß PACT02 wird dies durch einen Aufbau der PAE verhindert, bei welchem mindestens ein Register zur Entkopplung, etwa fest in den PAEs, definiert ist.

Generell sind unverzögerte Rückkopplungen durch Graphenanalyse des entstandenen kombinatorischen Netzes feststellbar. In die Datenpfade, in denen eine unverzögerte Rückkopplung besteht, werden daraufhin gezielt Register zur Entkopplung eingefügt. Der Compiler kann somit Register- beziehungsweise Speichermittel verwalten.

Durch die Verwendung von Handshake-Protokollen (z.B. RDY/ACK gem. 2.2.7) ist die korrekte Funktion der Berechnung auch durch das Einfügen von Registern sichergestellt.

2.5 Prozssormodell / Time Domain Multiplexing (TDM)

Grundsätzlich besitzt jede praktisch realisierte PAE-Matrix lediglich eine endliche Größe. Daher muß im folgenden Schritt eine Partitionierung des Algorithmus nach 2.2.5 Abs. 4 a/b in eine Mehrzahl von Konfigurationen durchgeführt werden, die nacheinander in die PAE-Matrix konfiguriert werden. Ziel ist typisch, möglichst viele Datenpakete in dem Netzwerk zu berechnen, ohne umkonfigurieren zu müssen.

Zwischen den Konfigurationen wird eine Zwischenspeicherung vorgenommen, wobei der Zwischenspeicher - ähnlich eines Registers bei CPUs - die Daten zwischen den einzelnen sequentiell ausgeführten Konfigurationen speichert.

Somit wird durch das Rekonfigurieren von Konfigurationen, der Datenverarbeitung in der PAE-Matrix und der Zwischenspeicherung in den Speichern ein sequentielles Prozessormodell aufgebaut.
Mit anderen Worten wird in der VPU-Technologie durch die beschriebene Compilierung nicht ein OpCode sequentiell ausgeführt, sondern komplexe Konfigurationen. Während bei CPUs ein Opcode typischerweise ein Datenwort bearbeitet, werden in der VPU-Technologie eine Mehrzahl von Datenworten (ein Datenpaket) von einer Konfiguration bearbeitet. Dadurch steigt die Effizienz der rekonfigurierbaren Architektur durch ein besseres Verhältnis zwischen Rekonfigurationsaufwand und Datenverarbeitung.

In der VPU-Technologie wird zugleich anstatt eines Registers ein Speicher verwendet, da nicht Datenworte, sondern Datenpakete zwischen den Konfigurationen bearbeitet werden.

Dieser Speicher kann als Random-Access Memory, Stack, FIFO oder als beliebige andere Speicherarchitektur ausgestaltet sein, wobei typischerweise mit einem FIFO die beste und am einfachsten zu realisiernde Möglichkeit gegeben ist.

Daten werden nunmehr durch die PAE-Matrix, entsprechend des konfigurierten Algorithmus, bearbeitet und in einem oder mehreren Speichern gespeichert. Die PAE-Matrix wird nach der Bearbeitung einer Menge von Daten umkonfiguriert und die neue Konfiguration entnimmt die Zwischenergebnisse aus dem/den Speicher(n) und setzt die Ausführung des Programmes fort. Da-

bei können durchaus auch neue Daten von externen Speichern und/oder der Peripherie zusätzlich in die Berechnung einfließen, ebenso können Ergebnisse an externen Speichern und/oder der Peripherie geschrieben werden.

Mit anderen Worten ist der typische Ablauf einer Datenverarbeitung das Auslesen von internen RAMs, das Verarbeiten der Daten in der Matrix und das Schreiben von Daten in die internen Speicher, wobei zur Datenverarbeitung auch beliebige externe Quellen oder Ziele für Datentransfers zusätzlich oder anstelle der internen Speicher verwendet werden können.

Während "sequencen" bei CPUs als das Neuladen eines OpCodes definiert ist, wird nach dem Vorstehenden "sequencen" von VPUs also als das (Re)konfigurieren von Konfigurationen definiert. Dies bedeutet allerdings nicht, daß nicht unter bestimmten Bedingungen Teile des Feldes als Sequenzer im herkömmlichen Sinne betrieben werden könnten.

Die Information, wann und/oder wie gesequenzt wird, d.h. welche nächste Konfiguration konfiguriert werden soll, ist durch verschiedene Informationen darstellbar, die einzeln oder kombiniert verwendet werden können. Z. B. sind folgende Strategien zur Ableitung der Information allein und/oder in Kombination bzw. alternativ sinnvoll:

a) durch den Compiler zur Übersetzungszeit definiert;
b) durch das Event-Netzwerk definiert
(Trigger, DE 197 04 728.9),
wobei das Eventnetzwerk interne und/oder externe Zustände repräsentieren kann;
c) durch den Füllgrad der Speicher
(Trigger, DE 197 04 728.9, DE 196 54 846.2-53).

2.5.1 Einfluß des TDM auf das Prozessormodell

Die Partitionierung des Algorithmus bestimmt entscheidend die relevanten Zustände, die in den Speichern zwischen den verschiedenen Konfigurationen abgelegt werden. Sofern ein Zustand nur innerhalb einer Konfiguration relevant ist (lokal relevanter Zustand), ist es nicht notwendig, diesen zu speichern, was vom Compilierverfahren bevorzugt berücksichtigt wird.

Zu Zwecken des Debuggings des auszuführenden Programmes kann es aber sinnvoll sein, diese Zustände dennoch zu speichern, um dem Debugger einen Zugriff auf diese zu ermöglichen. Auf die DE 101 42 904.5 wird verwiesen; diese ist hiermit vollumfänglich zu Offenbarungszwecken eingegliedert.

Weiterhin können zusätzlich Zustände relevant werden, wenn ein Taskswitch-Mechanismus (z.B. durch ein Betriebssystem oder Interruptquellen) verwendet wird und aktuelle ausgeführte Konfigurationen unterbrochen werden, andere Konfigurationen geladen werden und/oder zu einem späteren Zeitpunkt die abgebrochene Konfiguration fortgesetzt werden soll. Eine detailliertere Beschreibung folgt im nachfolgenden Abschnitt.

Ein einfaches Beispiel soll ein Unterscheidungsmerkmal für lokal relevante Zustände aufzeigen:

a) Eine Verzweigung des Types "if () then ... else ..." paßt vollständig in eine einzige Konfiguration, d.h. beide Datenpfade (Zweige) sind gemeinsam vollständig innerhalb der Konfiguration abgebildet. Der sich beim Vergleich ergebende Zustand ist relevant, jedoch lokal, da er in den nachfolgenden Konfigurationen nicht mehr benötigt wird.
b) Dieselbe Verzweigung ist zu groß, um vollständig in eine einzige Konfiguration zu passen. Mehrere Konfigurationen sind notwendig, um die vollständigen Datenpfade abzubilden. In diesem Fall ist der Zustand global relevant und muß gespeichert und den jeweiligen Daten zugeordnet werden, da die nachfolgenden Konfigurationen bei der Weiterverarbei-

tung der Daten den jeweiligen Zustand des Vergleichs benötigen.

2.6 Task-Switching

Einen zusätzlichen Einfluß auf die Betrachtung und den Umgang mit Zuständen hat der mögliche Einsatz eines Betriebssystemes. Betriebssysteme verwenden beispielsweise Task-Scheduler zum Verwalten mehrere Aufgaben (Tasks), um ein Multitasking zur Verfügung zu stellen.
Task-Scheduler brechen Tasks zu einem bestimmten Zeitpunkt ab, starten andere Tasks und kehren nach deren Abarbeitung zur Weiterbearbeitung des abgebrochenen Tasks zurück.
Sofern sichergestellt ist, daß eine Konfiguration - die hier der Abarbeitung eines Tasks entsprechen kann - nur nach der kompletten Abarbeitung - d.h. wenn alle innerhalb dieses Konfigurationszyklusses zu bearbeitende Daten und Zustände gespeichert sind - terminiert, können lokal relevante Zustände ungespeichert bleiben. Dieses Verfahren, also das komplette abarbeiten einer Konfiguration und der nachfolgende Taskswitch ist die bevorzugte Methode für den Betrieb von rekonfigurierbaren Prozessoren und entspricht im Wesentlichen dem Ablauf in einem normalen Prozessor, der auch zunächst die aktuell bearbeiteten Instruktionen abarbeitet und dann den Task wechselt.

Für manche Anwendungen ist jedoch eine besonders kurze Reaktion auf eine Taskwechselsanforderung erforderlich, z.B. in Realtime-Anwendungen. Hier kann es sinnvoll sein Konfigurationen vor deren kompletter Abarbeitung abzubrechen.
Sofern der Task-Scheduler Konfigurationen vor deren vollständiger Abarbeitung abbricht, müssen lokale Zustände und/oder Daten gespeichert werden. Weiterhin ist dies von Vorteil, wenn die Abarbeitungszeit einer Konfiguration nicht vorhergesagt werden kann. In Verbindung mit dem bekannten Halteproblem und dem Risiko, daß eine Konfiguration (z.B. durch einen Fehler)

gar nicht terminiert, erscheint dies weiterhin sinnvoll, um damit einen Deadlock des gesamten Systems zu verhindern.
Daher sind vorliegend, unter Berücksichtigung von Taskwechseln, relevante Zustände auch als solche anzusehen, die für einen Taskwechsel und ein erneutes korrektes Aufsetzen der Datenverarbeitung notwendig sind.
Bei einem Taskswitch ist somit der Speicher für Ergebnisse und ggf. auch der Speicher für die Operanden zu sichern und zu einem späteren Zeitpunkt, also bei der Rückkehr zu diesem Task, wieder herzustellen. Dies kann vergleichbar zu den PUSH/POP Befehlen und Verfahren nach dem Stand der Technik erfolgen.
Weiterhin ist der Zustand der Datenverarbeitung zu sichern, also der Zeiger auf die zuletzt vollständig bearbeiteten Operanden. Es sei hier besonders auf PACT18 verwiesen.

Abhängig von der Optimierung des Taskswitches gibt es beispielsweise zwei Möglichkeiten:

a) Die abgebrochene Konfiguration wird neu konfiguriert und nur die Operanden werden geladen. Die Datenverarbeitung beginnt so von neuem, als ob die Bearbeitung der Konfiguration noch gar nicht begonnen wurde. Mit anderen Worten werden einfach alle Datenberechnungen von vorne an ausgeführt, wobei ggf. Berechnungen bereits zuvor durchgeführt wurden. Diese Möglichkeit ist einfach, aber nicht effizient.

b) Die abgebrochene Konfiguration wird neu konfiguriert, wobei die Operanden und die bereits berechneten Ergebnisse in die jeweiligen Speicher geladen werden. Die Datenverarbeitung wird bei den Operanden fortgesetzt, die nicht mehr vollständig berechnet wurden. Dieses Verfahren ist sehr viel effizienter, setzt aber voraus, daß ggf. zusätzliche Zustände, die während der Verarbeitung der Konfiguration entstehen, relevant werden, etwa wenn zumindest ein Zeiger auf die zuletzt vollständig verrechneten Operanden gesichert

werden muss, damit bei deren Nachfolgern nach erfolgter neuer Konfiguration neu aufgesetzt werden kann.

2.7 Kontext Switch

Eine besonders bevorzugte Variante zur Verwaltung von relevanten Daten wird durch den nachfolgend beschriebenen Kontext Switch zur Verfügung gestellt. Bei Task-Wechseln und/oder bei der Ausführung von Konfigurationen und derem Wechsel (siehe beispielsweise Patentanmeldung DE 102 06 653.1, die zu Offenbarungszwecken vollumfänglich eingegliedert ist) kann es erforderlich sein, Daten oder Zustände, die typischerweise nicht zusammen mit den Arbeitsdaten in die Speicher abgelegt werden, da sie beispielsweise lediglich einen Endwert markieren, für eine nachfolgende Konfiguration zusichern.

Der erfindungsgemäß bevorzugt implementierte Kontext Switch wird derart durchgeführt, dass eine erste Konfiguration entfernt wird und die zu sichernden Daten in entsprechenden Speichern (REG) (Speicher, Register, Zähler, etc) verbleiben.

Dann kann eine zweite Konfiguration geladen werden, diese verbindet die REG in geeigneter Weise und definierter Reihenfolge mit einem oder mehreren globalen Speicher(n).
Die Konfiguration kann beispielsweise Adressgeneratoren verwenden, um auf den/die globalen Speicher zuzugreifen. Es ist also nicht erforderlich, vorab jeden einzelnen Speicherplatz durch den Compiler festlegen zu lassen und/oder auf als Speicher ausgestaltete REG zuzugreifen.
Entsprechend der konfigurierten Verbindung zwischen den REG werden die Inhalte der REG in einer definierten Reihenfolge in den globalen Speicher geschrieben, wobei die jeweiligen Adressen von Adressgeneratoren vorgegeben werden. Der Adressgenerator generiert die Adressen für den/die globalen Speicher(n) derart, dass die beschriebenen Speicherbereiche (PUSHAREA) der

entfernten ersten Konfiguration eindeutig zugeordnet werden können.
Es werden somit bevorzugt für unterschiedliche Konfigurationen unterschiedliche Adressenräume vorgesehen. Die Konfiguration entspricht dabei einem PUSH gewöhnlicher Prozessoren.

Danach verwenden andere Konfigurationen die Ressourcen.

Nun soll die erste Konfiguration wieder gestartet werden. Zuvor wird eine dritte Konfiguration gestartet, die die REG der ersten Konfiguration in einer definierten Reihenfolge miteinander verbindet.

Die Konfiguration kann wiederum beispielsweise Adressgeneratoren verwenden um auf den oder die globalen Speichern zuzugreifen und/oder um auf als Speicher ausgestaltete REG zuzugreifen.

Ein Adressgenerator generiert dabei Adressen bevorzugt derart, dass ein korrekter Zugriff auf die der ersten Konfiguration zugeordnete PUSHAREA erfolgt. Die generierten Adressen und die konfigurierte Reihenfolge der REG sind derart, dass die Daten der REG in der ursprünglichen Ordnung aus den Speichern in die REG geschrieben werden. Die Konfiguration entspricht einem POP gewöhnlicher Prozessoren.

Nun wird die erste Konfiguration wieder gestartet.

Zusammengefaßt wird ein Kontext Switch bevorzugt derart durchgeführt, dass durch das Laden besonderer Konfigurationen, die ähnlich von PUSH/POP bekannter Prozessorarchitekturen arbeiten, die zu sichernden Daten mit einem globalen Speicher ausgetauscht werden. Dieser Datenaustausch über globale Speicher mittels von Push/Pop-Austauschkonfiguationen wird als besonders relevant angesehen.

Die Funktion soll in einem Beispiel verdeutlicht werden:
Eine Funktion addiert 2 Zahlenreihen, die Länge der Reihen ist zur Übersetzungszeit nicht bekannt, sondern erst zur Laufzeit.

```
proc example
  while i<length do
    x[i] = a[i] + b[i]
    i = i + 1
```

Die Funktion wird nun während ihrer Ausführung unterbrochen, beispielsweise durch einen Task-Switch, oder weil der für x vorgesehene Speicher voll ist. a,b,x befinden sich zu diesem Zeitpunkt erfindungsgemäß in Speichern. i und ggf. length müssen jedoch gesichert werden.
Dazu wird die Konfiguration example terminiert, wobei die Registerinhalte erhalten bleiben und eine Konfiguration push gestartet, die i und length aus den Registern liest und in einen Speicher schreibt.

```
proc push
  mem[<push_adr_example>] = i
  push_adr_example++
  mem[<push_adr_example>] = length
```

Nach der Ausführung wird push terminiert und die Registerinhalte können gelöscht werden.
Andere Konfigurationen werden ausgeführt. Nach einiger Zeit wird die Konfiguration example wieder gestartet.
Zuvor wird eine Konfiguration pop gestartet, die die Registerinhalte wieder aus dem Speicher liest.

```
proc pop
  i = mem[<push_adr_example>]
```

```
push_adr_example++
length = mem[<push_adr_example>]
```

Nach der Ausführung wird pop terminiert und die Registerinhalte bleiben bestehen. Die Konfiguration example wird wieder gestartet.

2.8 Algorithmische Optimierung

Durch das beschriebene Übersetzungsverfahren werden Kontrollstrukturen von algorithmischen Strukturen getrennt. Beispielsweise zerfällt eine Schleife in einen Rumpf (WHILE) und eine algorithmische Struktur (Anweisungen).
Die algorithmischen Strukturen lassen sich nunmehr bevorzugt optional durch ein zusätzliches, der Trennung nachgeschaltetes Werkzeug optimieren.
Beispielsweise kann eine nachgeschaltetes Algebra-Software die programmierten Algorithmen optimieren und minimieren. Derartige Tools sind z.B. unter Bezeichnungen wie AXIOM, MARBLE, etc. bekannt. Durch die Minimierung kann eine schnellere Ausführung des Algorithmusses und/oder ein erheblich verringerter Platzbedarf erreicht werden.
Das Ergebnis der Optimierung wird danach wieder in den Compiler geführt und entsprechend weiterverarbeitet.
Es soll zudem angemerkt sein, dass moderne Compiler (-Frontends) bereits eine Anzahl von Optimierungen für Algorithemen (auch z.T. algebraische) implementiert haben, die selbstverständlich im Rahmen des hier beschriebenen Verfahrens weiterhin nutzbar sind.

Es soll ausdrücklich erwähnt sein, dass die beschriebenen Verfahren, insbesondere jedoch die Abschnitte 2.2.7 "Umgang mit Zeit" und 2.3 "Makros" auch auf Compiler nach PACT20 angewendet werden können. PACT20 wird diesbezüglich zu Offenbarungszwecken vollumfänglich in diese Patentanmeldung einbezogen.

3. Anwendbarkeit für Prozessoren nach dem Stand der Technik, insbesondere mit VLIW-Architektur

Es soll besonders angemerkt werden, daß anstatt einer PAE-Matrix auch eine Anordnung von arithmetisch logischen Einheiten nach dem Stand der Technik (ALUs), wie beispielsweise in VLIW-Prozessoren üblich, und/oder eine Anordnung von kompletten Prozessoren, wie beispielsweise in Multiprozessorsystemen üblich, verwendet werden kann. Ein Sonderfall stellt dabei die Verwendung einer einzelnen ALU das, sodaß das Verfahren auch für normale CPUs verwendbar ist.

In der Dissertation [Referenz Dissertation Armin Nückel] wurde ein Verfahren entwickelt, das die Übersetzung der WHILE-Sprache in semantisch korrekte endliche Automaten ermöglicht. Darüber hinaus kann ein endlicher Automat als "Unterprogramm" verwendet werden und umgekehrt. Dadurch entsteht die Möglichkeit, eine Konfiguration auf unterschiedliche Implementierungstechnologien abzubilden, wie z.B. CPUs; symmetrische Multiprozessoren; FPGAs; ASICs; VPUs.

Insbesondere ist es möglich, Teilen einer Applikation die jeweils optimal geeignete Hardware zuzuordnen bzw. eine jeweilie Eignung zu bestimmen und anhand der mehr oder weniger guten Eignung die optimale Hardware zuzuordnen. Dabei sind bevorzugt auch temporäre Ressourcenverteilungen und -reservierungen erfaßbar. Mit anderen Worten würde beispielsweise eine Datenflußstruktur einer Datenflußarchitektur zugeordnet werden, während eine sequentielle Struktur auf einen Sequenzer abgebildet wird, sofern diese vorhanden und/oder verfügbar sind.

Die entstehende Problemstellungen der Ressourcenzuweisungen für die einzelnen Algorithmen können z.B. durch einen "Job As-

signment"-Algorithmus zur Verwaltung der Zuordnung gelöst werden.

4. Implementierung

Die Implementierung eines erfindungsgemäßen Compilers soll von eine "normalen" sequentiellen Programmiersprache ausgehen, also z.B. C oder Pascal. Diese Sprachen weisen die Eigenschaft auf, dass durch ihren sequentiellen Charakter eine zeitliche Abfolge implizit und künstlich durch die Sprachendefinition an sich generiert wird.

Beispiel A:

```
Zeile 1:    i++
Zeile 2:    a = i * b
Zeile 3:    x = p - a
Zeile 4:    j = i * i
```

Durch die Sprachdefinition ist fest vorgegeben, dass Zeile 1 vor Zeile 2 vor Zeile 3 vor Zeile 4 ausgeführt wird. Allerdings könnte Zeile 4 auch direkt nach Zeile 1 ausgeführt werden und somit parallel zu Zeile 2 und 3 bearbeitet werden.

Mit anderen Worten werden durch sequentielle Sprachen weitere künstliche und nicht algorithmisch bedingte Zustände eingebaut. Wichtig ist lediglich die korrekte zeitliche Abfolge der Berechungen in Beispiel A. Zeile 4 darf erst berechnet werden, wenn i korrekt definiert ist, also nach der Abarbeitung von Zeile 1. Auch Zeile 2 darf erst nach der korrekten Definition von i (also nach der Abarbeitung von Zeile 1) verarbeitet werden. Zeile 3 benötigt die Ergebnisse von Zeile 2 (die Variable a) und darf daher erst nach derer korrekten Definition berechnet werden. Somit ergibt sich eine Datenabhängigkeit aber keine besonderen Zustände.

Anhand der Datenabhängigkeiten der Variable a in Zeile 2 und 3 (Zeile 3 verwendet a als Operand, a ist das Ergebnis von Zeile

2) kann automatisch durch den Compiler folgende Transformation zur Repräsentation der Parallelisier- bzw. Vektorisierbarkeit (ParVec-Transformation) durchgeführt werden:

```
Zeile 2:    VEC{a = i * b;
Zeile 3:        x = p - a}
```

VEC bedeutet, dass jeder durch ';' getrennte Ausdruck nacheinander abgearbeitet wird, wobei die Ausdrücke innerhalb der geschweiften Klammern grundsätzlich gepipelinet werden können. Bevorzugt müssen sämtliche Berechnungen am Ende von VEC{} durchgeführt und abgeschlossen sein, damit die Datenverarbeitung hinter VEC fortgesetzt wird.

Besser wird in einer internen Repräsentation der Datenstrukturen im Compiler die beiden Berechnungen als ein Vektor markiert:

```
VEC{a=i*b; x=p-q}
```

Zeile 4 ergibt einen einfachen Vektor:

```
VEC{j = i*i}
```

Da sich Zeile 4 gleichzeitig zu Zeile 2 und 3 berechnen lässt kann die Parallelität folgendermassen ausgedrückt werden:

```
PAR{{VEC{a=i*b; x=p-a};VEC{j=i*i}}
```

PAR bedeutet, dass jeder durch '{..}' getrennte Ausdruck zeitgleich abgearbeitet werden kann. Bevorzugt müssen sämtliche Berechnungen am Ende von PAR{} durchgeführt und abgeschlossen sein, damit die Datenverarbeitung hinter PAR fortgesetzt wird.

Wird Zeile 1 mit einbezogen, ergibt sich:

```
VEC{i++; PAR{{VEC{a=i*b; x=p-a}}{VEC{j=i*i}}}}
```

Da VEC{j=i*i} ein Vektor mit nur einem Element darstellt, kann auch wir folgt geschrieben werden:

```
VEC{i++; PAR{{VEC{a=i*b; x=p-a}}{j=i*i}}}
```

Ein weiteres Beispiel zeigt einen echten Zustand.

Beispiel B:

```
Zeile 1:    i++
Zeile 2:    a = i * b
Zeile 3:    if a < 100 {
Zeile 4:        x = p - a
Zeile 5:    } else {
Zeile 6:        j = i * i }
```

Jetzt kann Zeile 6 nur noch nach der Berechnung von Zeile 2 und Zeile 3 ausgeführt werden. Die Berechnung von Zeile 4 und 6 findet alternativ statt. Also ist der Zustand von Zeile 3 für die weitere Datenverarbeitung relevant (relevanter Zustand).

Bedingte Zustände können bei einer Transformation durch IF ausgedrückt werden:

```
Zeile 1-2:  VEC{i++; a=i*b}
Zeile 3:    IF{{a<100}{zeile4}{zeile6}}
Zeile 4:    VEC{x=p-a}
Zeile 6:    VEC{j=i*i}
```

Zusammengefaßt ergibt das

```
VEC{i++; a=i*b; IF{{a<100}{VEC{x=p-a}}{VEC{j=i*i}}}}
```

Weitere relevante Zustände werden durch Schleifen erzeugt:
Beispiel C:

```
Zeile 1:    for (i = 1, i < 100, i++)
Zeile 2:        a = a * i
Zeile 3:    q = p / a
```

Zeile 3 darf erst ausgeführt werden, nachdem die Schleife terminiert ist. Also bestehen bei bedingten Sprüngen relevante Zustände.

Eine erste Transformation der Schleife ergibt:

```
Zeile 1:        i=1;
Zeile 2: loop:  if i >= 100 then exit
Zeile 3:        a = a * i
Zeile 4:        i++
Zeile 5:        jump loop
Zeile 6: exit:  q = p / a
```

Zeile 3 und 4 können parallel berechnet werden, da Zeile 4 nicht vom Ergebnis von Zeile 3 abhängt:

```
PAR{{a=a*i}{i++}}
```

Zeile 5 ergibt einen Vektor mit dem generierten PAR, da erst nach vollständiger Berechnung der Werte wieder in die Schleife gesprungen werden darf (hier liegt also eine zeitliche Abhängigkeit vor).

```
VEC{PAR{{a=a*i}{i++}}; jump loop}
```

Somit ergibt sich für die Bedingung:

```
loop: IF{{i>=100}{jump exit}{VEC{PAR{{a=a*i}{i++}}; jump
loop}}}
```

Die Zeile 1 ist ein Vektor mit der Bedingung, da diese vor der Bedingung ausgeführt werden muss (IF verwendet i als Operand, i ist das Ergebnis von Zeile 1).
Zeile 6 ist wiederum ein Vektor mit der Bedingung, da a als Operand verwendet wird und a das Ergebnis der Bedingung ist.

Somit ergibt sich (in übersichtlicher Schreibweise):

```
VEC{
    i++;
    loop: IF{
        {i>=100}
        {jump exit}
        {VEC{
            PAR{
                {a=a*i}
                {i++}
            };
            jump loop
            }
        }
    };
    exit: q=p/a
}
```

Die Inhalte von VEC{} und PAR{} können als rein kombinatorische Netze betrachtet werden.
Bevorzugt wird VEC und PAR ale Petri-Netz ausgestaltet, um wie bevorzugt die Weiterverarbeitung nach kompletter Verarbeitung der jeweilgen Inhalte zu steuern.

Durch die mögliche Betrachtung von VEC und PAR als rein kombinatorisches Netz entsteht die Notwendigkeit den Schleifenzustand zu sichern. D.h. in diesem Fall ist es tatsächlich notwendig einen endlichen Automaten zu schaffen.

Die Anweisung REG{} speichert dazu Variablen in einem Register. Somit entsteht durch die Verwendung der kombinatorischen Netze VEC und PAC in Verbindung mit dem Register REG ein endlicher Automat, der exakt entsprechend des Algorithmus aufgebaut ist:

```
VEC{
    i++;
    loop: IF{
        {i>=100}
        {jump exit}
        {VEC{
            PAR{
                {a=a*i}
                {i++}
            };
            REG{a;i}
            jump loop
            }
        }
    };
    exit: q=p/a
}
```

Es soll besonders darauf hingewiesen werden, dass in der VPU Technologie des Anmelders (vgl. PACT21) Ein- und/oder Ausgangsregister an den PAEs vorgesehen sind und die zeitliche Korrektheit und die Verfügbarkeit von Daten durch ein integriertes Handshake-Protokoll (RDY/ACK) sichergestellt ist. Insoweit wird die Forderung bevorzugt beim Verlassen von VEC{} oder PAR{} deren interne Datenverarbeitung abgeschlossen zu haben automatisch für alle nachfolgend verwendeten Veriablen erfüllt (wäre die Datenverarbeitung nicht beendet, würden nachfolgende Berechnungsschritte auf die Beendigung und das

Eintreffen der Daten warten). Durch die integrierten Register sind auch schwingende Rückkopplungen ausgeschlossen.

Insoweit ist nachfolgender Term für diese Technologie korrekt:
VEC{PAR{{a=a*i}{i++}}; jump loop}

Für andere Technologien, die die o.g. Ausgestaltungen nicht oder nur teilweise aufweisen, sollte der Term folgendermassen formuliert werden:
VEC{PAR{{a=a*i}{i++}}; REG{a;i}; jump loop}

Es soll darauf hingewiesen werden, dass diese Form auf jeden Fall auch in der VPU-Technologie des Anmelders zu einer korrekten und optimalen Abbildung des Algorithmus auf den rekonfigurierbaren Prozessor führt.

REG kann innerhalb der kombinatorischen Netze VEC und PAR verwendet werden. Streng betrachtet verlieren dadurch VEC und PAR die Eigenschaft der kombinatorischen Netze. Abstrakt kann jedoch REG als ein komplexes Element (REG-Element) eines kombinatorischen Netzes betrachtet werden, dem eine eigene Abarbeitungszeit zugrunde liegt. Die Bearbeitung der nachfolgenden Elemente wird vom der Beendigung der Berechnung des REG-Elementes abhängig gemacht.
In dem Bewusstsein dieser begrifflichen Ungenauigkeit wird eine Verwendung von REG innerhalb von VEC und PAR im Weiteren zugelassen und ist insbesondere auch notwendig.

Wie bereits vorstehend erwähnt, ist die Verwendung von REG typischerweise innerhalb einer Konfiguration einer VPU des Anmelders nicht erforderlich, sondern explizit immer nur dann, wenn die Berechnungsergebnisse einer Konfiguration abgespeichert werden, sodass REG ein diesem Anwendungsfall tatsächlich dem expliziten Register eines endlichen Automaten entspricht.

Neben der Synthese von endlichen Automaten für Schleifen, sind insbesondere in einem weiteren Fall endliche Automaten erforderlich:

Ist ein Algorithmus zu groß, um komplett innerhalb der PAEs eines rekonfigurierbaren Prozessors abgearbeitet zu werden, muß er in mehrere Teilalgorithmen zerlegt werden. Jeder Teilalgorithmus stellt eine Konfiguration für den rekonfigurierbaren Prozessor dar. Nacheinander, also sequentiell, werden die Teilalgorithmen auf den Prozessor konfiguriert, wobei die Ergebnisse der jeweils vorhergehenden Konfiguration(en) für die jeweils neue Konfiguration als Operanden dienen.

Mit anderen Worten entsteht durch die Rekonfiguration ein endlicher Automat, der zu einem Zeitpunkt t Daten bearbeitet und speichert und zu einem Zeitpunkt t+1, möglicherweise nach einer Konfiguration, die gespeicherten Daten ggf. anders verarbeitet und wieder speichert. Wesentlich ist, dass t nicht im klassischen Sinn durch Takte oder Befehle definiert wird, sondern durch Konfigurationen. Hierzu sein besonders die Präsentation Prozessormodell (PACT, Oktober 2000, San Jose) referenziert.

Mit noch anderen Worten besteht eine Konfiguration aus einem kombinatorischen Netz aus VEC und/oder PAR, dessen Ergebnisse gespeichert werden (REG), um in der nächsten Konfiguration weiterverwendet zu werden:

```
Konfiguration 1:    VEC{Operands;{VEC|PAR};REG{Results1}}
Konfiguration 2:    VEC{Results1;{VEC|PAR};REG{Results2}}
...
```

Zum einfacheren Verständnis haben die obigen Beispiels und Beschreibungen die Konstrukte VEC, PAR und REG in der Hochsprachen eingeführt und diese dadurch strukturiert. Typischerweise

und bevorzugt wird diese Strukturierung erst aber auf der Ebene der Zwischensprache (siehe Principles of Compiler Design (Red Dragon), Aho, Sethi, Ullmann) eingeführt.
Es soll besonders darauf hingewiesen werden, dass die Strukturierung von Algorithmen mit VEC, PAR und REG typischerweise vollkommen automatisch durch den Compiler durch Methoden wie z.B. Graphenanalyse durchführbar ist.
Insbesondere ist es aber auch denkbar und teilweise von Vorteil dem Programmierer selbst die Strukturierungsmöglichkeit in der Hochsprache dadurch zu ermöglichen, dass VEC, PAR und REG wie oben aufgezeigt direkt in der Hochsprache beschreibbar sind.

Generierung

Die automatische Erstellung von VEC, PAR und REG kann auf unterschiedlichen Ebenen einen Compilierungsvorganges durchgeführt werden. Die zunächst einleuchtendste ist während eines Präprozessor-Durchlaufes auf Basis des Source-Codes wie in den vorigen Beispielen beschrieben. Für die weitere Compilierung ist danach allerdings ein speziell angepasster Compiler erforderlich.

Ein weiterer Aspekt ist, dass Compiler zumeist automatische Optimierungen von Code vornehmen (z.B. in Schleifen). Eine effiziente Zerlegung des Codes ist daher erst nach den Optimierungsläufen sinnvoll, insbesondere wenn Compiler (wie z.B. SUIF, Universität Stanford) bereits den Code für Parallelisierung und/oder Vectorisierung hin optimieren.

Die daher besonders bevorzugte Methode ist die Einbindung der Analysen in das Backend eines Compilers. Das Backend übersetzt eine compilerinterne Datenstruktur auf die Befehle eines Zielprozessors.

Als compilerinterne Datenstrukturen werden zumeist Zeigerstrukturen wie DAGs/GAGs, Trees oder 3-Adress-Codes verwendet (siehe Principles of Compiler Design (Red Dragon), Aho, Sethi, Ullmann). Teilweise werden auch Stack-Machine-Codes verwendet (siehe Compiler selbstgeschneidert, C'T 1986 1-5). Da die Datenformate prinzipiell äquivalent sind und ineinander transformiert werden können, setzt die erfindungsgemäß bevorzugte Methode auf der Weiterverarbeitung von Graphen, wie bevorzugt Trees, auf.

Datenabhängikeiten und mögliche Parallelitäten entsprechend dem vorstehend beschriebenen Verfahren sind innerhalb von Trees einfach auf Basis der Struktur automatisch zu erkennen. Hierzu können beispielsweise bekannte und etablierte Verfahren der Graphenanalyse eingesetzt werden. Alternativ oder optional kann durch entsprechend adaptierte Parsingmethoden ein Algorithmus auf Datenabhängikeiten, Schleifen, Sprünge etc. hin untersucht werden. Dabei kann ein Verfahren ähnlich dem der Auswertung von Ausdrücken in Compilern verwendet werden.

Abbildung

Die weitere Transformation des Algorithmus ist nunmehr stark von der Zielarchitektur abhängig. Beispielsweise bietet die Prozessorarchitektur des Anmelders (VPU, XPP) automatische Datensynchronisation in Hardware. Das bedeutet, dass die korrekte zeitliche Abfolge von Datenabhängigkeiten automatisch in der Hardware gehandhabt wird. Andere Architekturen benötigen zum Teil zusätzlich die Synthese geeigneter Zustandsmaschinen für die Steuerung es Datentransfers.

Besonders interessant ist die Handhabung bedingter Sprünge. Beispielswiese stellt die Prozessorarchitektur des Anmelders

mehrere Mechanismen zu derer Abbildung und Ausführung zur Verfügung:

1. Rekonfiguration des Prozessors oder Teilen des Prozessors durch eine übergeordnete Konfigurationseinheit (vgl. Patentanmeldung(en) PACT01, 04, 05, 10, 13, 17)
2. Auswalzen der Funktion in das Array aus PAEs (vgl. Patentanmeldung PACT08), dabei werden z.B. beide möglichen Zweige eines Vergleiches zugleich auf das Array abgebildet.
3. Wave Rekonfiguration nach Patentanmeldung(en) PACT08, 13, 17), dabei wird den unterschiedlich zu bearbeitenden Daten ein Token mitgegeben, das die jeweils gültige Konfiguration wählt.

Es soll erwähnt sein, dass der Mechanismus 1 der allgemein typisch anzuwendende Fall ist. Der Mechanismus 2 ist bereits bei den meisten Technologien sehr aufwendig oder gar nicht implementierbar und der Fall 3 ist bislang nur aus der VPU-Technologie des Anmelders bekannt.

Die jeweils zu wählende Ausführungsmethode hängt von der Komplexität des Algorithmus, dem erforderlichen Datendurchsatz (Performance) und der exakten Ausgestaltung des Zielprozessors ab (z.B. Anzahl der PAEs).
Beispiele:
Ein einfacher Vergleich soll folgendes Berechnen:
if i < 0 then a=a*(-i) else a=a*i

Eine Rekonfiguration des Prozessors (Mechanismus 1) je nach Ergebnis des Vergleichs scheint wenig sinnvoll zu sein.
Das Auswalzen beider Zweige in das Array (Mechanismus 2) ist Grundsätzlich möglich. Je nach Ergebnis des Vergleichs werden entweder die a=a*(-i) oder a=a*i berechnenden PAEs angesteuert (vgl. PACT08).
Besonders platzeffizient ist das Überlagern der beiden Berechnungen (Mechanismus 3), wodurch nach dem Vergleich unabhängig vom Ergebnis dieselbe(n) PAEs die Daten weiterverarbeiten, die

Daten aber mit einem Token versehen sind, das sodann in Abhängigkeit vom Vergleich lokal in den jeweils nachfolgenden die Daten verarbeitenden PAEs entweder die Funktion a=a*(-i) oder a=a*i auswählt. (vgl. PACT08, 13, 17).

Nach Mechanismus 1 entsteht ein global relevanter Zustand, da die komplette folgende Konfiguration davon abhängig ist.

Nach Mechanismus 2 und 3 entstehen nur ein lokal relevanter Zustand, da dieser über die Berechnung hinaus - die vollständig implementiert ist - nicht mehr benötigt wird.

Mit anderen Worten kann die lokale oder globale Relevanz von Zuständen auch von der gewählten Abbildung auf die Prozessorarchitektur abhängen.

Ein Zustand der über eine Konfiguration hinaus und somit über das kombinatorische Netz des eine Konfiguration repräsentierenden endlichen Automaten hinaus relevant ist (also von nachfolgenden endlichen Automaten benötigt wird), kann grundsätzlich als global betrachtet werden. Es soll nochmals auf die verwendete diffuse Terminologie des Begriffes kombinatorisches Netz hingewiesen werden.

Befehlsmodell des entstandenen Prozessors

Entsprechend der vorliegenden Erfindung entsteht ein Prozessormodell für rekonfigurierbare Prozessoren, das alle wesentlichen Befehle umfaßt:

Arithmetisch/logische Befehle werden direkt in das kombinatorische Netz abgebildet.
Sprünge (Jump/Call) werden entweder direkt in das kombinatorische Netz ausgewalzt oder durch Rekonfiguration realisiert.

Bedingung und Kontrollflußbefehle (if, etc) werden entweder im kombinatorischen Netz vollständig aufgelöst und bearbeitet oder an eine übergeordnete Konfigurationseinheit weitergeleitet, die sodann entsprechend des entstandenen Status eine Rekonfiguration durchführt.

Load/Store-Operationen werden bevorzugt in separate Konfigurationen abgebildet und durch Adressgeneratoren ähnlich den bekannten DMA's realisiert, die internen Speicher (REG{}) mittels Adressgeneratoren in externe Speicher schreiben oder diese von externen Speichern und/oder Peripherie laden. Sie können aber auch zusammen mit der datenverarbeitenden Konfiguration konfiguriert sein und arbeiten.

Register-Move-Operationen werden im kombinatorischen Netz durch Busse zwischen den internen Speichern (REG{}) realisiert.

Push/Pop-Operationen werden durch separate Konfigurationen realisiert, die ggf. bestimmte interne Register im kombinatorischen Netz und/oder die internen Speicher (REG{}) mittels Adressgeneratoren in externe Speicher schreiben oder aus externen Speichern lesen und die bevorzugt vor oder nach den eigentlichen datenverarbeitenden Konfigurationen ausgeführt werden.

5. Beschreibung der Figuren

Die nachfolgenden Figuren zeigen Implementierungs- und Ausgestaltungsbeispiele des Compilers.

Figur 1a zeigt den Aufbau eines gewöhnlichen endlichen Automaten, bei welchem ein kombinatorisches Netz (0101) mit einem Register (0102) verknüpft ist. Daten können direkt an 0101 (0103) und 0102 (0104) geführt werden. Durch eine Rückkopplung (0105) des Registers auf das kombinatorische Netz ist die Verarbeitung eines Zustandes in Abhängigkeit des/der vorhergehen-

den Zustände möglich. Die Verarbeitungsergebnisse werden durch 0106 dargestellt.

Figur 1b zeigt eine Repräsentation des endlichen Automaten durch eine rekonfigurierbare Architektur nach PACT01 und PACT04 (PACT04 Fig. 12-15). Das kombinatorischen Netz aus Figur 1a (0101) wird durch eine Anordnung von PAEs 0107 ersetzt (0101b). Das Register (0102) wird durch einen Speicher (0102b) ausgeführt, der mehrere Zyklen speichern kann. Die Rückkopplung gemäß 0105 erfolgt durch 0105b. Die Eingänge (0103b bzw. 0104b) sind äquivalent 0103 bzw 0104. Der direkte Zugriff auf 0102b kann ggf. durch einen Bus durch das Array 0101b realisiert werden. Der Ausgang 0106b ist wiederum äquivalent 0106.

Figur 2 zeigt die Abbildung eines endlichen Automaten auf eine rekonfigurierbare Architektur. 0201(x) repräsentieren das kombinatorische Netz (das entsprechend Figur 1b als PAEs ausgestaltet sein kann). Es existieren ein oder mehrere Speicher für Operanden (0202) und ein oder mehrere Speicher für Ergebnisse (0203). Zusätzliche Daten Ein-/Ausgänge gem. 0103b, 0104b, 0106b) sind der Einfachheit halber nicht dargestellt. Den Speichern zugeordnet ist jeweils ein Adressgenerator (0204, 0205).
Die Operanden- und Ergebnisspeicher (0202, 0203) sind physikalisch oder virtuell derart miteinander verkoppelt, daß beispielsweise die Ergebnisse einer Funktion bzw. einer Operation einer anderen als Operanden dienen können und/oder sowohl Ergebnisse als auch neu zugeführte Operanden einer Funktion einer anderen als Operanden dienen können. Eine derartige Kopplung kann beispielsweise durch Bussysteme hergestellt werden oder durch eine (Re)Konfiguration durch welche die Funktion und Vernetzung der Speicher mit den 0201 neu konfiguriert wird.

Figur 3 zeigt verschiedene Aspekte zum Umgang mit Variablen. In **Figur 3a** zeigen 0301, 0302, 0303 verschiedene Stufen der Berechnung. Diese Stufen können rein kombinatorisch oder auch über Register voneinander getrennt sein. f1, f2, f3 sind Funktionen, x1 ist eine Variable gemäß Patentbeschreibung.
Figur 3b zeigt die Verwendung einer Variablen x1 in der Funktion x1 := x1 + 1.
Figur 3c zeigt das Verhalten eines endlichen Automaten zur Berechnung von x1 := x1 + 1 innerhalb einer Konfiguration. In der nächsten Konfiguration sind 0306 und 0304 zu vertauschen um einen vollständigen endlichen Automaten zu erhalten. 0305 repräsentiert die Adressgeneratoren für die Speicher 0304 und 0306.

Figur 4 zeigt Implementierungen von Schleifen. Die schraffierten Module können durch Makros generiert werden (0420, 0421). 0421 kann auch durch Analyse des Graphen auf unverzögerte Rückkopplungen eingefügt werden.
Figur 4a zeigt die Implementierung einer einfachen Schleife der Art

```
WHILE TRUE DO
    x1 := x1 + 1;
```

Im Kern der Schleife liegt der Zähler +1 (0401). 0402 ist ein Multiplexer, der zu Beginn den Startwert von x1 (0403) auf 0401 führt und sodann bei jeder Iteration die Rückkopplung (0404a, 0404b) bewirkt. In die Rückkopplung ist ein Register (vgl. REG{}) (0405) eingesetzt, um eine unverzögerte und damit unkontrollierte Rückkopplung des Ausgangs von 0401 auf dessen Eingang zu verhindern. 0405 wird mit dem Arbeitstakt der VPU getaktet und bestimmt damit die Anzahl der Iterationen pro Zeit. Der jeweilige Zählerstand wäre an 0404a oder 0404b abgreifbar. Je nach Definition der Hochsprache terminiert die Schleife jedoch nicht. Beispielsweise wäre in einer HDL (nach dem Stand der Technik (z.B. VHDL, Verilog) das Signal auf 0404 nutzbar, während es in einer sequentiellen Programmiersprache

(z.B. C) 0404 nicht nutzbar ist, da die Schleife nicht terminiert und somit keinen Exit-Wert liefert.
Der Multiplexer 0402 realisiert ein Makro, das aus dem Schleifenkonstrukt entstanden ist. Das Makro wird durch die Übersetzung von WHILE instantiiert.
Das Register 0405 ist entweder ebenfalls Teil des Makros oder wird entsprechend einer Graphenanalyse nach dem Stand der Technik exakt dann und dort eingefügt, wo eine unverzögerte Rückkopplung existiert, um so die Schwingneigung auszuschalten.
Figur 4b zeigt den Aufbau einer echten Schleife der Art

```
WHILE x1 < 10 DO
    x1 := x1 + 1;
```

Der Aufbau entspricht im Kern der Figur 4a, weshalb dieselben Referenzen verwendet wurden.
Zusätzlich ist eine Schaltung dargestellt, die die Gültigkeit des Ergebnisses kontrolliert (0410) und das Signal von 0404a nur dann an die nachfolgenden Funktionen (0411) weiterleitet, wenn das Abbruchkriterium der Schleife erreicht ist. Das Abbruchkriterium wird durch den Vergleich x1 < 10 festgestellt (Vergleichsstufe 0412). Als Ergebnis des Vergleiches wird das betreffende Statusflag (0413) an ein Multipliziermittel 0402 zur Steuerung der Schleife und die Funktionen 0411 zur Kontrolle der Ergebnisweiterführung geleitet. Das Statusflag 0413 kann beispielsweise durch Trigger gemäß DE 197 04 728.9 implementiert sein. Ebenfalls kann das Statusflagmittel 0413 an eine CT gesendet werden, die daraufhin die Terminierung der Schleife erkennt und eine Rekonfiguration durchführt.

Figur 5a zeigt die iterative Berechnung von

```
FOR i:=1 TO 10
    x1 := x1 * x1;
```

Im wesentlichen entspricht die Grundfunktion Figur 4b, weshalb die Referenzen übernommen wurden. Der Funktionsblock 0501 berechnet die Multiplikation. Die FOR-Schleife wird durch eine

weitere Schleife entsprechend Figur 4b implementiert und ist lediglich durch Block 0503 angedeutet. Block 0503 liefert den Status des Vergleiches auf das Abbruchkriterium. Der Status wird direkt zur Ansteuerung der Iteration verwendet, wodurch das Mittel 0412 (dargestellt durch 0502) weitgehend entfällt.

Figur 5b zeigt das Auswalzen der Berechnung von

```
FOR i:=1 TO 10
    x1 := x1 * x1;
```

Da die Anzahl der Iterationen zur Übersetzungszeit exakt bekannt ist, kann die Berechnung in eine Folge von i Multiplizierern (0510) abgebildet werden.

Figur 6 zeigt die Ausführung einer WHILE-Schleife gem. Figur 4b über mehrere Konfigurationen. Hier ist der Zustand der Schleife (0601) ein relevanter Zustand, da dieser die Funktion in den nachfolgenden Konfigurationen maßgeblich beeinflußt. Die Berechnung erstreckt sich über 4 Konfigurationen (0602, 0603, 0604, 0605). Zwischen den Konfigurationen werden die Daten in Speichern (vgl. REG{}) abgelegt (0606, 0607). 0607 ersetzt dabei ebenfalls 0405.
Als ein Rekonfigurationskriterium kann der Füllstand der Speicher dienen, angedeutet über 0606, 0607: Speicher voll/leer und/oder 0601, das den Abbruch der Schleife anzeigt. Mit anderen Worten werden durch den Füllstand der Speicher Trigger generiert (vgl. PACT01, PACT05, PACT08, PACT10), die an die Konfigurationseinheit gesendet werden und eine Rekonfiguration auslösen. Auch der Zustand der Schleife (0601) kann an die CT gesendet werden. Daraufhin kann die CT bei Erreichen des Abbruchkriteriums die nachfolgenden Algorithmen konfigurieren, bzw. ggf. zunächst die restlichen Teile der Schleife (0603, 0604, 0605) abarbeiten und danach die nachfolgenden Konfigurationen laden.

6. Parallelisierbarkeit

Figur 6 zeigt potentielle Grenzen der Parallelisierbarkeit auf.

Sofern die Berechnung der Operanden unabhängig von der Rückkopplung 0608 ist, kann die Schleife blockweise, d.h. jeweils durch Füllen der Speicher 0606/0607 berechnet werden. Damit wird ein hoher Grad an Parallelität erreicht.

Sofern die Berechnung eines Operanden abhängig von dem Ergebnis der vorherigen Berechnung ist, also eine Rückkopplung oder dergleichen 0608 in die Berechnung einfließt, wird das Verfahren ineffizienter, da jeweils nur ein Operand innerhalb der Schleife berechnet werden kann.

Ist der nutzbare ILP (Instruktionslevel Parallelismus) innerhalb der Schleife hoch und die Zeit für die Rekonfiguration nieder (vgl. PACT02, PACT04, PACT13, PACT17), kann eine auf PAEs ausgewalzte Berechnung auf einer VPU weiterhin effizient sein.
Ist dies nicht der Fall, ist es sinnvoll, die Schleife auf eine sequentielle Architektur (vom PA separater Prozessor oder Implementierung innerhalb des PA entsprechend DE 196 51 075.9-53, DE 196 54 846.2-53 und insbesondere DE 199 26 538.0 (Fig. 5, 11, 16, 17, 23, 30, 31, 33)) abzubilden.

Die Analyse der Berechnungszeiten kann entweder im Compiler zur Übersetzungszeit gemäß dem nachfolgenden Abschnitt erfolgen und/oder empirisch zu der oder einer Laufzeit gemessen werden, um eine nachträgliche Optimierung herbeizuführen, was zu einem lernfähigen, insbesondere selbstlernenden Compiler führt.

Für die Erfindung sind Analyse- und Parallelisierungsverfahren von Bedeutung.
Verschiedene Verfahren nach dem Stand der Technik stehen für die Analyse und Durchführung der Parallelisierung zur Verfügung.
Ein bevorzugtes Verfahren soll im Folgenden beschrieben werden.
Abzubildende Funktionen werden durch Graphen dargestellt (vgl. PACT13; DE 199 26 538.0), wobei eine Applikation aus beliebig vielen unterschiedlichen Funktionen zusammengesetzt sein kann. Die Graphen werden auf die in ihnen enthaltene Parallelität untersucht, wobei vorab sämtliche Methoden der Optimierung zum Einsatz kommen können.
Beispielsweise sollen folgende Untersuchungen durchgeführt werden:

6.0.1 ILP (Instruction Level Parallelism)

ILP drückt aus, welche Befehle zeitgleich ausgeführt werden können (vgl. PAR{}). Eine derartige Analyse wird auf Basis der Betrachtung von Abhängigkeiten von Knoten in einem Graphen einfach möglich. Entsprechende Verfahren sind nach dem Stand der Technik und in der Mathematik per se hinreichend bekannt, es soll beispielsweise auf VLIW-Compiler und Synthesetools verwiesen werden.

Besondere Beachtung benötigen aber z. B. gegebenenfalls verschachtelte bedingte Ausführungen (IF), da eine korrekte Aussage der parallel ausführbaren Pfade oftmals kaum oder nicht zu treffen ist, da eine starke Abhängigkeit vom Werteraum der einzelnen Parameter besteht, der oftmals nicht oder nur unzureichend bekannt ist. Auch kann eine exakte Analyse derart viel Rechenzeit in Anspruch nehmen, daß sie nicht mehr sinnvoll durchführbar ist.

In derartigen Fällen kann beispielsweise die Analyse durch Hinweise vom Programmierer vereinfacht werden und/oder es kann anhand entsprechender Compilerschalter derart gearbeitet werden, daß im Zweifelsfall entweder von einer hohen Parallelisierbarkeit (ggf. unter Verschwendung von Ressourcen) oder von einer niederen Parallelisierbarkeit (ggf. unter Verschwendung von Performance) ausgegangen werden soll.
Ebenfalls kann in diesen Fällen eine empirische Analyse zur Laufzeit durchgeführt werden. Nach PACT10, PACT17 sind Verfahren bekannt, die zur Laufzeit die Erstellung von Statistiken über das Programmverhalten erlauben. Derart kann z. B. zunächst von einer maximalen Parallelisierbarkeit ausgegangen werden. Die einzelnen Pfade geben Meldungen an eine Statistikeinheit (z. B. implementiert in einer CT oder einer anderen Stufe,vgl. PACT10, PACT17, grundsätzlich können aber auch Einheiten nach PACT04 verwenden werden) über jeden Durchlauf zurück. Mittels statistischer Maßnahmen ist nunmehr auswertbar, welche Pfade tatsächlich parallel durchlaufen werden. Weiterhin ergibt sich die Möglichkeit, anhand der Daten zur Laufzeit zu bewerten, welche Pfade häufig oder selten oder nie parallel durchlaufen werden. Diese Art der Pfadnutzungsmeldung ist nicht zwingend, aber vorteilhaft.
Dementsprechend kann die Ausführung bei einem nächsten Programmaufruf optimiert werden. Daß dazu die Statistikinformation insbesondere nichtflüchtig wie auf eine Festplatte weggeschrieben werden kann, sei erwähnt. Aus PACT22, PACT24 ist bekannt, daß mehrere Konfigurationen entweder zugleich konfiguriert werden können und danach durch Trigger (PACT08) angesteuert werden oder nur eine Untermenge konfiguriert ist und die restlichen Konfigurationen bei Bedarf dadurch nachgeladen werden, daß die entsprechenden Trigger an eine Ladeeinheit (CT, PACT10) gesendet werden.

Der im folgenden gebrauchte Wert PAR(p) gibt zur Verdeutlichung an, welche Parallelität auf Instruktionsniveau, d. h.

wieviel ILP bei einer bestimmten Stufe (p) innerhalb des aus der Funktion transformierten Datenflußgraphen erreichbar ist (Figur 7a).

Gleichfalls bedeutsam ist Vektorparallelität (vgl. VEC{}). Vektorparallelität ist nutzbar, wenn größere Datenmengen zu verarbeiten sind. In diesem Fall sind lineare Folgen von Operationen vektorisierbar, d.h. alle Operationen können gleichzeitig Datenverarbeiten, wobei typischerweise jede separate Operation ein separates Datenwort bearbeitet.

Innerhalb von Schleifen ist dieses Vorgehen teilweise nicht möglich. Daher sind Analysen und Optimierungen notwendig. Beispielsweise kann der Graph einer Funktion durch ein Petrinetz ausgedrückt werden. Petri-Netze besitzen die Eigenschaft, daß die Ergebnisweitergabe von Knoten kontrolliert erfolgt, wodurch beispielsweise Schleifen modelliert werden können. Durch die Rückkopplung des Ergebnisses in einer Schleife wird der Datendurchsatz bestimmt. Beispiele:

- Das Ergebnis der Berechnung n wird für die Berechnung n+1 benötigt: Nur eine Berechnung kann innerhalb der Schleife ausgeführt werden.
- Das Ergebnis der Berechnung n wird für die Berechnung n+m benötigt: m-1 Berechnungen können innerhalb der Schleife ausgeführt werden.
- Das Ergebnis bestimmt den Abbruch der Schleife, geht aber nicht in die Berechnung der Ergebnisse ein: Eine Rückkopplung ist nicht notwendig. Ggf. laufen zwar falsche (zuviel) Werte in die Schleife, jedoch kann die Ausgabe der Ergebnisse direkt bei Erreichen der Endbedingung am Schleifenende unterbrochen werden.

Vor der Analyse von Schleifen können diese nach dem Stand der Technik optimiert werden. Beispielsweise können bestimmte In-

struktionen aus der Schleife herausgezogen werden und vor oder nach die Schleife gestellt werden.

Der im Folgenden zur Verdeutlichung gebrauchte Wert VEC kann den Grad der Vektorisierbarkeit einer Funktion veranschaulichen. Mit anderen Worten zeigt VEC an, wieviele Datenworte zugleich innerhalb einer Menge von Operationen bearbeitet werden können. VEC kann beispielsweise aus der Zahl der benötigten Rechenwerke für eine Funktion n_{nodes} und der zugleich innerhalb des Vektors berechenbaren Daten n_{data} berechnet werden, z.B. durch $VEC = n_{nodes} / n_{data}$

Ist eine Funktion beispielsweise auf 5 Rechenwerke abbildbar ($n_{nodes} = 5$) und in jedem der Rechenwerke können zugleich Daten bearbeitet werden ($n_{data} = 5$) ist VEC = 1 (Figur 7b).
Ist eine Funktion dagegen beispielsweise auf 5 Rechenwerke abbildbar ($n_{nodes} = 5$) und nur in einem Rechenwerk können jeweils Daten bearbeitet werden, z. B. aufgrund einer Rückkopplung der Ergebnisse der Pipeline auf den Eingang ($n_{data} = 5$), so ist VEC = 1/5 (Figur 7c).

VEC kann für eine gesamte Funktion und/oder für Teilausschnitte einer Funktion berechnet werden. Für den erfindungsgemäßen Compiler können beide Varianten vorteilhaft sein, wie generell die Bestimmung und Auswertung von VEC vorteilhaft ist.

Gemäß Figur 7a wird PAR(p) für jede Zeile eines Graphen bestimmt, wie vorteilhaft möglich. Eine Zeile eines Graphen ist dadurch definiert, daß sie innerhalb einer Takteinheit ausgeführt wird. Die Anzahl der Operationen ist von der Implementierung der jeweiligen VPU abhängig.

Entspricht PAR(p) der Anzahl der Knoten in der Zeile p, so können alle Knoten parallel ausgeführt werden.

Ist PAR(p) kleiner, werden bestimmte Knoten nur alternativ ausgeführt. Die alternativen Ausführungen jeweils eines Knotens werden in jeweils einer PAE zusammengefaßt. Eine Selektionsvorrichtung ermöglicht die Aktivierung der, dem Status der Datenverarbeitung entsprechenden, Alternative zur Laufzeit wie beispielsweise in PACT08 beschrieben.

VEC wird ebenfalls jeder Zeile eines Graphen zugeordnet. Ist für eine Zeile VEC = 1, bedeutet dies, daß die Zeile als Pipelinestufe bestehen bleibt. Ist eine Zeile kleiner 1, so werden alle nachfolgenden Zeilen, die ebenfalls kleiner 1 sind zusammengefaßt, da ein Pipelining nicht möglich ist. Entsprechend der Reihenfolge der Operationen werden diese zu einer Sequenz zusammengefaßt, die dann in eine PAE konfiguriert wird und zur Laufzeit sequentiell abgearbeitet wird. Entsprechende Verfahren sind beispielsweise aus PCT/DE 97/02949 und/oder PCT/DE 97/02998 bekannt.

Durch das beschriebene Verfahren lassen sich durch Gruppierungen von Sequenzern beliebig komplexe Parallelprozessormodelle aufbauen. Insbesondere sind Sequenzerstrukturen zur Abbildung von reentrantem Code generierbar.
Die dazu jeweils notwendigen Synchronisationen können beispielsweise durch das in PACT18 beschriebene TimeStamp-Verfahren oder bevorzugt durch das in PACT08 beschriebene Triggerverfahren durchgeführt werden.

Werden mehrere Sequenzer oder sequentielle Teile auf ein PA abgebildet, ist es aus Leistungsverbrauchsgründen bevorzugt, die Leistung der einzelnen Sequenzer aufeinander abzustimmen. Dies kann besonders bevorzugt derart geschehen, daß die Arbeitsfrequenzen der Sequenzer aneinander angepaßt werden. Aus PACT25 und PACT18 sind beispielsweise Verfahren bekannt, die eine individuelle Taktung von einzelnen PAEs oder PAE-Gruppen zulassen.

Die Frequenz eines Sequenzers kann dabei anhand der Anzahl von Zyklen bestimmt werden, die er typischerweise zur Abarbeitung der ihm zugewiesenen Funktion benötigt.
Benötigt er beispielsweise 5 Taktzyklen zur Abarbeitung seiner Funktion während das restliche System genau einen Taktzyklus benötigt, um zugewiesene Aufgaben abzuarbeiten, sollte seine Taktung 5-mal höher sein als die Taktung des restlichen Systems. Bei einer Vielzahl von Sequenzern sind jeweils unterschiedliche Taktzyklen möglich. Es kann eine Taktvervielfachung und/oder eine Taktteilung vorgesehen werden.

Funktionen werden entsprechend des vorgenannten Verfahrens partitioniert. Beim Partitionieren werden entsprechend Speicher für Daten und relevanten Status eingefügt. Weitere alternative und/oder weitergehende Verfahren sind aus PACT13 und PACT18 bekannt.
Manche VPUs bieten nach PACT01, PACT10, PACT13, PACT17, PACT22, PACT24 die Möglichkeit der differentiellen Rekonfiguration. Diese kann angewendet werden, wenn nur verhältnismäßig wenige Änderungen innerhalb der Anordnung von PAEs bei einer Rekonfiguration notwendig werden. Mit anderen Worten werden nur die Veränderungen einer Konfiguration gegenüber der aktuellen Konfiguration rekonfiguriert. Die Partitionierung kann in diesem Fall dergestalt sein, daß die auf eine Konfiguration folgenden (differentielle) Konfiguration nur die notwendigen Rekonfigurationsdaten enthält und keine vollständige Konfiguration darstellt. Der Compiler der vorliegenden Erfindung ist bevorzugt dazu ausgebildet, dies zu erkennen und zu unterstützen.

Das Scheduling der Rekonfiguration kann durch den Status erfolgen, der Funktion(en) an eine Ladeeinheit (CT) meldet, welche ihrerseits auf Basis des eingehenden Status die nächste Konfiguration oder Teilkonfiguration auswählt und konfigu-

riert. Im Detail sind derartige Verfahren aus PACT01, PACT05, PACT10, PACT13, PACT17 bekannt.
Weiterhin kann das Scheduling die Möglichkeit des Vorladens von Konfigurationen während der Laufzeit einer anderen Konfiguration unterstützen. Dabei können mehrere Konfigurationen möglicherweise auch spekulativ vorgeladen werden, d.h. ohne daß sichergestellt ist, daß die Konfigurationen überhaupt benötigt werden. Dies ist besonders dann bevorzugt, wenn die CT etwa bei längeren, konfigurationsfrei abarbeitbaren Datenströmen zumindest weitgehend unbelastet ist und insbesondere nicht oder nur wenig aufgabenbelastet ist. Durch Selektionsmechanismen wie etwa nach DE 197 04 728.9 werden dann zur Laufzeit die zu verwendenden Konfigurationen ausgewählt (siehe auch Beispiel NLS in PACT22/24).

Ebenfalls können die lokalen Sequenzer durch den Status ihrer Datenverarbeitung gesteuert werden, wie etwa aus DE 196 51 075.9-53, DE 196 54 846.2-53, DE 199 26 538.0 bekannt. Zur Durchführung ihrer Rekonfiguration kann ein weiterer abhängiger oder unabhängiger Status an die CT gemeldet werden (siehe beispielsweise PACT04, LLBACK).

Das Vorstehende wird nun mit Bezug auf weitere Figuren beschrieben. Dabei werden im folgenden folgende Zeichen zur Vereinfachung der Schreibung verwendet: $\vee$ oder, $\wedge$ und.

Figur 8a zeigt die Abbildung des Graphens nach Fig. 7a auf eine Gruppe von PAEs bei maximaler erreichbarer Parallelität. Sämtliche Operationen (Instruktion i1-i12) sind in einzelne PAEs abgebildet.

Figur 8b zeigt denselben Graphen, beispielsweise mit maximaler nutzbarer Vektorisierbarkeit. Jedoch sind die Mengen von Operationen V2={i1, i3}, V3={i4, i5, i6, i7, i8}, V4={i9, i10, i11} nicht parallel par (par({2,3,4})= 1. Damit lassen sich

Ressourcen einsparen, indem jeweils eine Menge P2, P3, P4 von Operationen einer PAE zugeordnet wird. Ein Statussignal zu jedem Datenwort in jeder Stufe wählt die auszuführende Operation in der jeweiligen PAE aus. Die PAEs sind als Pipeline (Vektor) vernetzt und jede PAE führt je Takt eine Operation über jeweils unterschiedliche Datenwort aus.

Es ergibt sich folgender Ablauf:
PAE1 berechnet Daten und gibt diese an PAE2 weiter. Zusammen mit den Daten gibt sie ein Statussignal weiter, das anzeigt, ob i1 oder i2 ausgeführt werden soll.
PAE2 berechnet die Daten von PAE1 weiter. Entsprechend des eingehenden Statussignals wird die auszuführende Operation (i1, i2) ausgewählt und berechnet. Entsprechend der Berechnung gibt PAE2 ein Statussignal an PAE3 weiter, das anzeigt, ob (i4 v i5) v (i6 v i7 v i8) ausgeführt werden soll.
PAE3 berechnet die Daten von PAE2 weiter. Entsprechend des eingehenden Statussignals wird die auszuführende Operation (i4 v i5) v (i6 v i7 v i8) ausgewählt und berechnet. Entsprechend der Berechnung gibt PAE3 ein Statussignal an PAE4 weiter, das anzeigt ob i9 v i10 v i11 ausgeführt werden soll.
PAE4 berechnet die Daten von PAE3 weiter. Entsprechend des eingehenden Statussignals wird die auszuführende Operation i9 v i10 v i11 ausgewählt und berechnet.
PAE5 berechnet die Daten von PAE4 weiter.

Ein mögliches entsprechendes Verfahren und Hardware, die eine besonders günstige Umsetzung des beschriebenen erlaubt, ist in DE 197 04 728.9 (Figuren 5 und 6) beschrieben; auch PACT04 und PACT10, PACT13 beschreiben allgemein nutzbare, jedoch aufwendigere Verfahren.

Figur 8c zeigt wiederum denselben Graphen. In diesem Beispiel ist eine Vektorisierung nicht möglich, jedoch ist PAR(p) hoch,

was bedeutet, daß innerhalb einer Zeile jeweils eine Vielzahl von Operationen gleichzeitig ausgeführt werden kann. Die parallel durchführbaren Operationen sind P2 = {i1 ∧ i2}, P3 = {i4 ∧ i5 ∧ i6 ∧ i7 ∧ i8}, P4 = {i9 ∧ i10 ∧ i11}. Die PAEs sind derart vernetzt, daß sie beliebige Daten beliebig untereinander austauschen können. Die einzelnen PAEs führen nur dann Operationen durch, wenn im entsprechenden Zyklus ein ILP besteht, ansonsten verhalten sie sich neutral (NOP), wobei ggf. Heruntertaktung und/oder eine Takt- und/oder Stromabschaltung zur Minimierung der Verlustleistung erfolgen kann.
Es ist dabei folgender Ablauf vorgesehen:
Im *ersten* Zyklus arbeitet nur PAE2 und gibt die Daten an PAE2 und PAE3 weiter.
Im *zweiten* Zyklus arbeiten PAE2 und PAE3 parallel und geben ihre Daten an PAE1, PAE2, PAE3, PAE4, PAE5 weiter.
Im *dritten* Zyklus arbeiten PAE1, PAE2, PAE3, PAE4, PAE5 und geben die Daten an PAE2, PAE3, PAE5 weiter.
Im *vierten* Zyklus arbeiten PAE2, PAE3, PAE5 und geben die Daten an PAE2 weiter.
Im *fünften* Zyklus arbeitet nur PAE2.

Die Funktion benötigt somit 5 Zyklen zur Berechnung. Der entsprechende Sequenzer sollte also mit dem 5-fachen Takt im Verhältnis zu seiner Umgebung arbeiten, um eine entsprechende Performance zu erzielen.

Ein mögliches entsprechendes Verfahren ist in PACT02 (Figuren 19, 20 und 21) beschrieben; auch PACT04 und PACT10, 13 beschreiben allgemein nutzbare, jedoch aufwendigere Verfahren. Weitere Verfahren und/oder Hardware sind verwendbar.

Figur 8d zeigt den Graphen nach Fig. 7a für den Fall, daß keinerlei nutzbare Parallelität besteht. Zur Berechnung eines Datenwortes muß jede Stufe nacheinander durchlaufen werden. In-

nerhalb der Stufen wird immer nur genau einer der Zweige verarbeitet.
Die Funktion benötigt ebenfalls 5 Zyklen zur Berechnung, cy1 = (i1), cy2 = (i2 ∨ i3), cy3 = (i4 ∨ i5 ∨ i6 ∨ i7 ∨ i8), cy4 = (i9 ∨ i10 ∨ i11), cy5 = (i12) . Der entsprechende Sequenzer sollte also mit dem 5-fachen Takt im Verhältnis zu seiner Umgebung arbeiten, um eine entsprechende Performance zu erzielen.
Eine derartige Funktion ist beispielsweise ähnlich Fig. 8c durch einen einfachen Sequenzer nach PACT02 (Figuren 19, 20 und 21) abbildbar. Auch PACT04 und PACT10, 13 beschreiben allgemein nutzbare, jedoch aufwendigere Verfahren.

Die in Figur 8 dargestellten Abbildungen sind beliebig mischbar und gruppierbar.
In **Figur 9a** ist beispielsweise dieselbe Funktion dargestellt, bei welcher die Pfade (i2 ∧ (i4 ∨ i5) ∧ i9) und (i3 ∧ (i6 ∨ i7 ∨ i8) ∧ (i9 ∨ i10)) parallel ausführbar sind. (i4 ∨ i5), i6 ∨ i7 ∨ i8), (i9 ∨ i10) sind jeweils alternativ. Die Funktion ist weiterhin vektorisierbar. Damit läst sich eine Pipeline aufbauen, in welcher für 3 PAEs (PAE4, PAE5, PAE7) jeweils anhand von Statussignalen die jeweilig auszuführende Funktion bestimmt ist.

Figur 9b zeigt ein ähnliches Beispiel, bei dem eine Vektorisierung nicht möglich ist. Allerdings sind die Pfade (i1 ∧ i2 ∧ (i4 ∨ i5) ∧ i9 ∧ i12) und (i3 ∧ (i6 ∨ i7 ∨ i8) ∧ (i10 ∨ i11)) parallel. Damit läßt sich die optimale Performance durch den Einsatz von zwei PAEs erzielen, die bei die parallelen Pfade auch parallel abarbeiten. Die Synchronisation der PAEs untereinander erfolgt durch Statussignale, die vorzugsweise von PAE1 generiert werden, da diese den Beginn (i1) und das Ende (i12) der Funktion berechnet.

Es soll besonders darauf hingewiesen werden, daß sich aus einer mehrfachen Anordnung von Sequenzern ein symetrisch paralleles Prozessormodell (SMP) oder ähnliche, heute verwendete Mehrprozessormodelle ergeben können.

Weiterhin soll darauf hingewiesen werden, daß sämtliche Konfigurationsregister für das Scheduling auch im Hintergrund und/oder während der Datenverarbeitung mit neuen Konfigurationen geladen werden können.

Es ist dies etwa möglich, wenn die Hardware wie nach DE 196 51 075.9-53 bekannt aufgebaut ist. Es stehen dann unabhängige Speicherbereiche oder Register zur Verfügung, die unabhängig angesprochen werden können. Auf bestimmte Stellen wird durch eingehende Trigger gesprungen, ebenfalls kann mittels Sprungbefehlen (JMP, CALL/RET), die ggf. auch bedingt durchführbar sind gesprungen werden.

Gemäß DE 196 54 846.2-53 stehen unabhängige Schreib- und Lesezeiger zur Verfügung, wodurch grundsätzlich eine Unabhängigkeit und somit die Möglichkeit des Zugriffes im Hintergrund gegeben ist. Insbesondere ist es möglich, die Speicher zu segmentieren, wodurch eine zusätzliche Unabhängigkeit gegeben ist. Mittels Sprungbefehlen (JMP, CALL/RET), die ggf. auch bedingt durchführbar sind, kann gesprungen werden.

Nach DE 197 04 728.9 sind die einzelnen Register, die durch die Trigger gewählt werden können, grundsätzlich unabhängig und erlauben daher eine unabhängige Konfiguration, insbesondere im Hintergrund. Sprünge innerhalb der Register sind nicht möglich, die Auswahl erfolgt ausschließlich über die Triggervektoren.

Ein wesentlicher Faktor zur Bewertung der Effizienz von PAR und VEC ist die Art der Daten die durch die jeweilige Struktur verarbeitet werden. Beispielsweise ist es lohnen eine Struktur auszuwalzen, also zu pipelinen und oder parallelisieren, die eine große Menge von Daten verarbeitet; wie es z.B. bei Videodaten oder Telekomdaten der Fall ist. Strukturen die wenige Daten verarbeiten (z.B. Tastatureingabe, Maus, etc.) lohnen sich nicht ausgewalzt zu werden, im Gegenteil sie würden nur anderen Algorithmen die Ressourcen blockieren.

Somit wird vorgeschlagen anhand unterschiedlicher Hinweise nur die Algorithmen, Strukturen oder Teile von Algorithmen zu parallelisiern und vektorisieren, die entsprechend große Datenmengen Verarbeiten.
Derartige Hinweise können beispielsweise sein:
1. Der Datentyp (Arrays, Streams sollten z.B. aufgrund der hohen Datenmenge eher ausgewalzt werden als z.B. einzelne Zeichen).
2. Die Art des Zugriffes (lineare Programmabfolgen sollten z.B.in Sequenzer abgebildet werden, während Schleifen sich z.B. aufgrund der hohen Anzahl von Durchläufen zum Auswalzen lohnen.
3. Die Art der Quelle und/oder des Ziels (Tastatur und Maus haben z.B. eine zu geringe Datenrate um effizient ausgewalzt zu werden, dagegen ist z.B. die Datenrate bei Netzwerk und/oder Video Quellen oder Zielen deutlich höher).

Für die Analyse können dabei eine beliebige Menge dieser Hinweise hinzugezogen werden.

7. Begriffsdefinition

lokal relevanter Zustand	Zustand, der nur innerhalb einer bestimmten Konfiguration relevant ist;
global relevanter Zustand	Zustand, der in mehreren Konfigurationen relevant ist und zwischen den Konfigurationen ausgetauscht werden muß;
relevanter Zustand	Zustand, der innerhalb eines Algorithmus zu dessen korrekter Ausführung dessen benötigt wird und somit durch den Algorithmus beschrieben ist und davon verwendet wird;
irrelevanter Zustand	Zustand, der für den eigentlichen Algorithmus ohne Bedeutung ist und auch nicht im Algorithmus beschrieben ist, der jedoch von der ausführenden Hardware implementierungsabhängig benötigt wird

Patentansprüche

1. Verfahren zum Übersetzen von Hochsprachen auf rekonfigurierbare Architekturen, dadurch gekennzeichnet, daß ein endlicher Automat zur Berechnung derart aufgebaut wird, daß ein komplexes kombinatorisches Netz aus einzelnen Funktionen gebildet wird und dem Netz Speicher zur Speicherung der Operanden und Ergebnissen zugeordnet sind.

2. Verfahren zum Datenbe- und/oder verarbeitung mit einem multidimensionalen Feld mit rekonfigurierbaren ALUs, dadurch gekennzeichnet, daß ein Hochsprachencode vorgesehen und derart übersetzt wird, daß ein endlicher Automat zur Berechnung aufgebaut wird, wobei ein komplexes kombinatorisches Netz aus einzelnen Funktionen gebildet und dem Netz Speicher zur Speicherung der Operanden und/oder Ergebnisse zugeordnet werden.

3. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß das komplexe kombinatorische Netz so aufgebaut und/oder zerlegt wird, daß die PAE-Matrix möglichst lange ohne Rekonfiguration betrieben wird.

4. Verfahren nach dem vorhergehenden Anspruch, dadurch gekennzeichnet, daß komplexe Instruktionen bestimmt werden, um das komplexe kombinatorische Netz so aufzubauen und/oder zu zerlegen, daß die PAE-Matrix möglichst lange ohne Rekonfiguration betrieben wird.

5. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein endlicher Automat direkt aus imperativem Quelltext aufgebaut wird.

6. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein endlicher Automat aus an grobgranulare Logikkreise und/oder an vorhandene feingranulare Elemente (FPGA-Zellen in der VPU, statemachines etc.) angepaßte Operationen aufgebaut wird, insbesondere ausschließlich an solche.

7. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein endlicher Automat dann in Konfigurationen zerlegt wird.

8. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß generierte Konfigurationen succesive auf die PAE-Matrix abgebildet werden und Arbeitsdaten und/oder Zustände, die zwischen den Konfigurationen zu übertragen sind, in Speicher abgelegt werden.

9. Verfahren nach dem vorhergehenden Anspruch, dadurch gekennzeichnet, daß der Speicher vom Compiler bestimmt beziehungsweise vorgesehen wird.

10. Verfahren nach dem vorhergehenden Anspruch, dadurch gekennzeichnet, daß während einer Konfiguration Daten aus einer VPU externen Quelle und/oder einem internen Speicher verarbeitet und an eine externe Quelle und/oder einen internen Speicher geschrieben werden.

11. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein Speicher für einen gesamten Datensatz vorgesehen wird, der umfangreicher als ein einzelnes Datenwort ist.

12. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß bei Verarbeitung einer ablaufenden Kon-

figuration Daten compilerbestimmt in die Speicher abgelegt werden.

13. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein Speicher für Operanden, ein Speicher für Ergebnisse und ein Netzwerk aus Zuweisungen und/oder Vergleichen-Anweisungen, also Bedingungen wie z.B. IF, CASE, Schleifen (WHILE, FOR, REPEAT) sowie optionalen Adressgenerator(en) zur Ansteuerung der Speicher mit dem Automaten vorgesehen werden.

14. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß Zuständen wie erforderlich Speicher zugeordnet werden, und hierbei zwischen algorithmisch relevanten und irrelevanten Zuständen unterschieden wird, insbesondere solchen relevanten Zuständen, die innerhalb des Algorithmus notwendig um dessen korrekte Funktion zu beschreiben und solchen irrelevante Zustände, die durch die verwendete Hardware und/oder die gewählte Abbildung oder aus anderen sekundären Gründen entstehen.

15. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß Load/Store Operationen unter Vorsehen einer externen Adressierung, also des Datentransfers mit externen Baugruppen und einer interne Adressierung, also die Datentransfers zwischen PAEs, i.b. zwischen RAM-PAEs und ALU-PAEs vorgesehen werden.

16. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß bei der Datenverarbeitung eine erste Konfiguration entfernt wird und die zu sichernden Daten in entsprechenden Speichern (REG) (Speicher, Register, Zähler, etc) verbleiben.

17. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß die erste Konfiguration wieder geladen wird und auf die zuvor gesicherterten, ihr zugeordnete Daten zugreift.

18. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß für den Zugriff auf zuvor gesicherterte Daten eine zweite Konfiguration geladen wird, die die REG in geeigneter Weise und definierter Reihenfolge mit einem oder mehreren globalen Speicher(n) verbindet, insbesondere, um unter Verwendung von Adressgeneratoren auf den/die globalen Speicher zuzugreifen, wobei der Adressgenerator die Adressen für den/die globalen Speicher(n) bevorzugt derart generiert, dass die beschriebenen Speicherbereiche (PUSHAREA) der entfernten ersten Konfiguration eindeutig zugeordnet werden können.

19. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß automatisch Transformation zur Repräsentation der Parallelisier- bzw. Vektorisierbarkeit (Par-Vec-Transformation) durchgeführt werden und/oder VEC und PAR-Anteile als Petri-Netz ausgestaltet werden, um wie bevorzugt die Weiterverarbeitung nach kompletter Verarbeitung der jeweiligen Inhalte zu steuern.

20. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß
arithmetisch/logische Befehle direkt in das kombinatorische Netz abgebildet werden und/oder
Sprünge (Jump/Call) entweder direkt in das kombinatorische Netz ausgewalzt und/oder durch Rekonfiguration realisiert werden und/oder
Bedingungen und Kontrollflußbefehle (if, etc) entweder im kombinatorischen Netz vollständig aufgelöst und /oder bearbeitet werden und/oder an eine übergeordnete Konfigurati-

onseinheit weitergeleitet werden, die sodann entsprechend des entstandenen Status eine Rekonfiguration durchführt und/oder

Load/Store-Operationen in separate Konfigurationen abgebildet und/oder durch Adressgeneratoren realisiert werden, die internen Speicher (REG{}) mittels Adressgeneratoren in externe Speicher schreiben und/oder diese von externen Speichern und/oder Peripherie laden und/oder

Register-Move-Operationen im kombinatorischen Netz durch Busse zwischen den internen Speichern (REG{}) realisiert werden und/oder

Push/Pop-Operationen durch separate Konfigurationen realisiert werden, die bestimmte interne Register im kombinatorischen Netz und/oder die internen Speicher (REG{}) mittels Adressgeneratoren in externe Speicher schreiben oder aus externen Speichern lesen und die bevorzugt vor oder nach den eigentlichen datenverarbeitenden Konfigurationen ausgeführt werden.

Fig. 1a

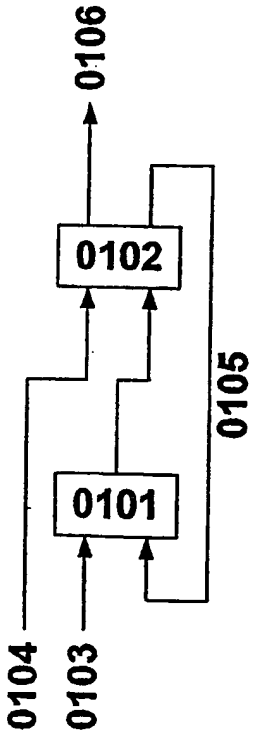
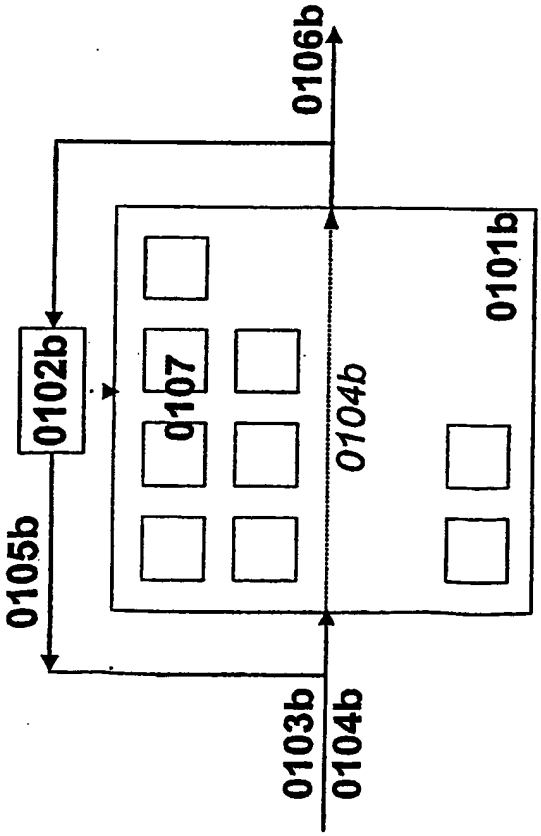

0106
0102
0105
0101
0104
0103

Fig. 1b


0106b
0101b
0102b
0107
0104b
0105b
0103b
0104b

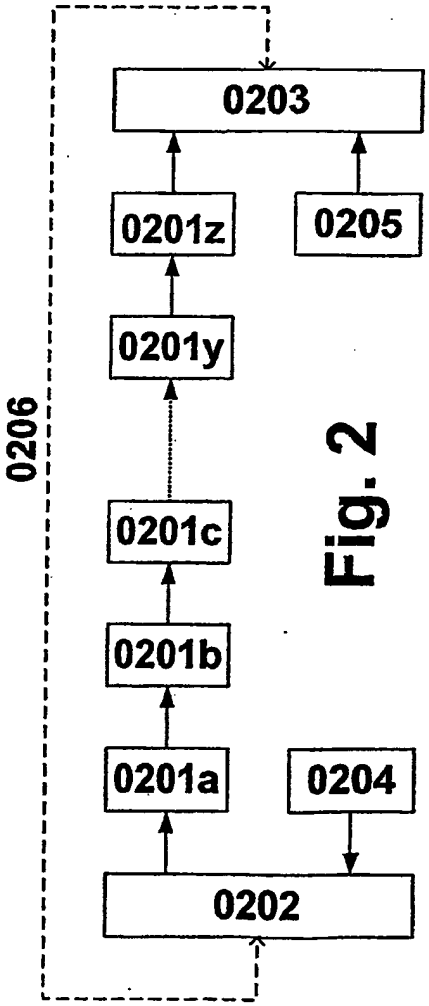
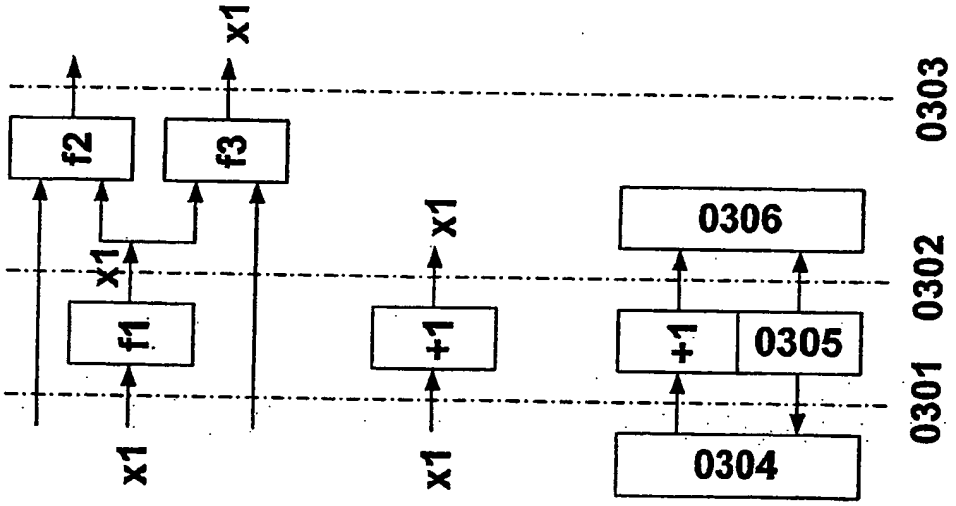


Fig. 2

Fig. 3a

Fig. 3b

Fig. 3c



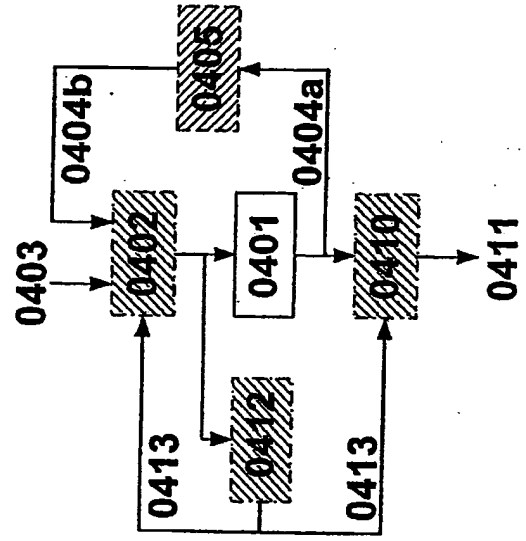

0403
0402
0404b
0405
0401
0404a
0410
0411
0413
0412
0413

Fig. 4b

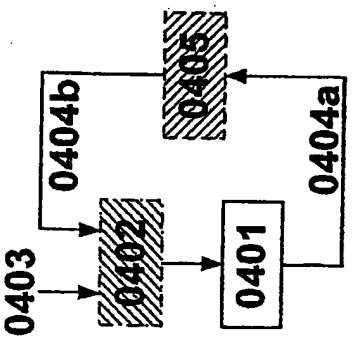
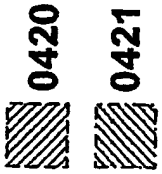

0403
0402
0404b
0405
0401
0404a

Fig. 4a


0420
0421

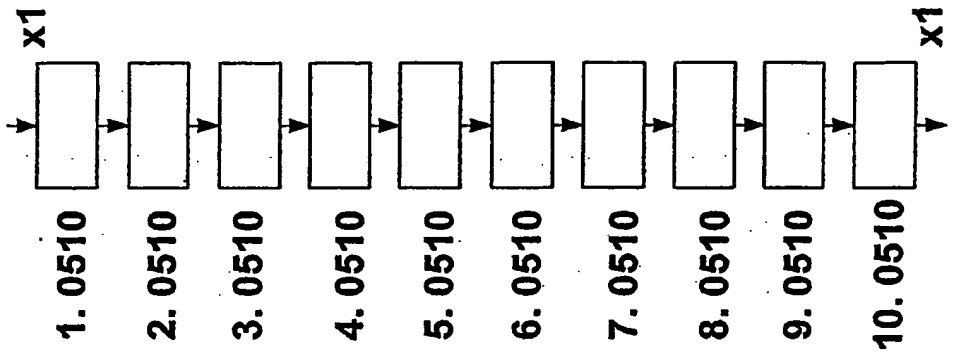

x1
1. 0510
2. 0510
3. 0510
4. 0510
5. 0510
6. 0510
7. 0510
8. 0510
9. 0510
10. 0510
x1

Fig. 5b

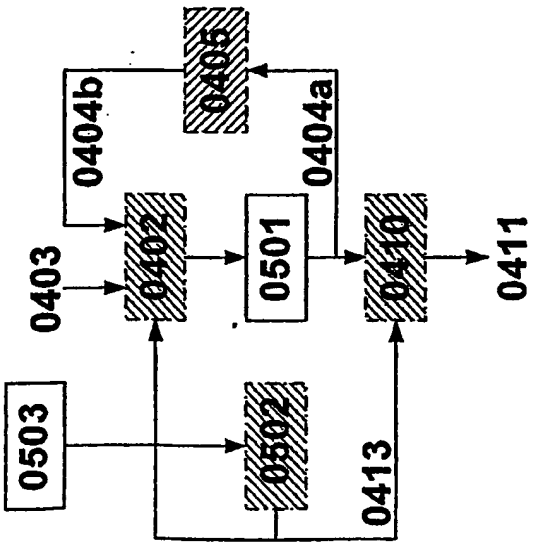

0503
0403
0402
0404b
0405
0404a
0501
0410
0411
0502
0413

Fig. 5a

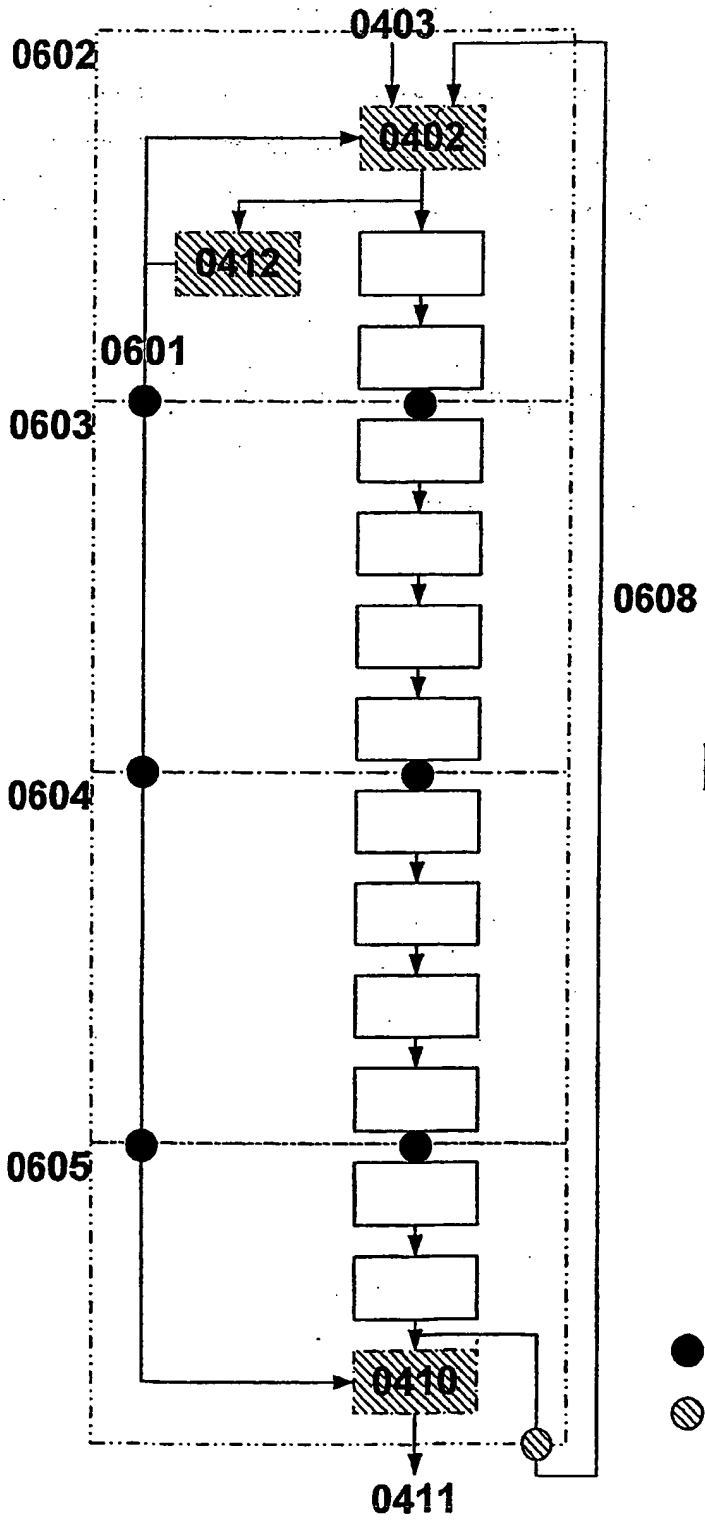

0403
0602
0402
0412
0601
0603
0608
0604
0605
0410
0411
0606
0607 (0405)

Fig. 6

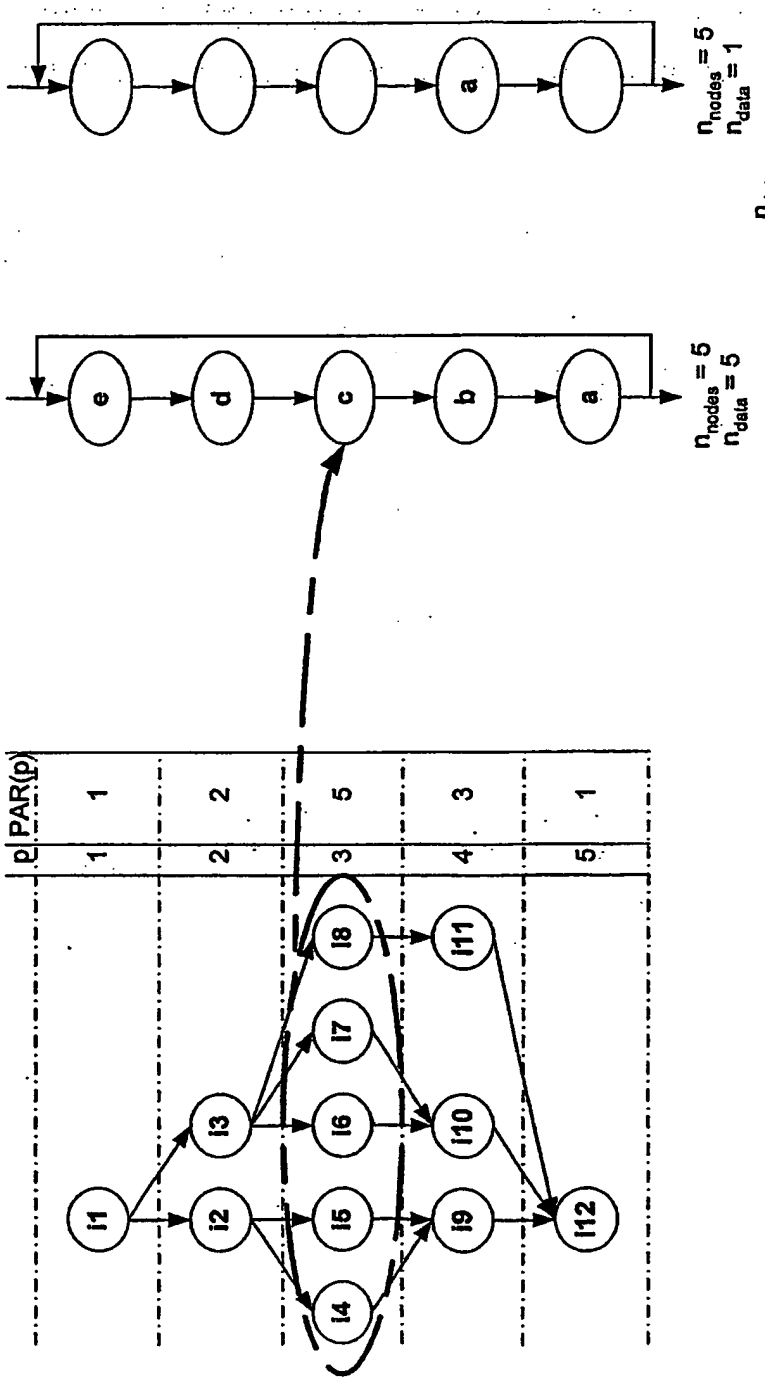


Fig. 7a

Fig. 7b

Fig. 7c

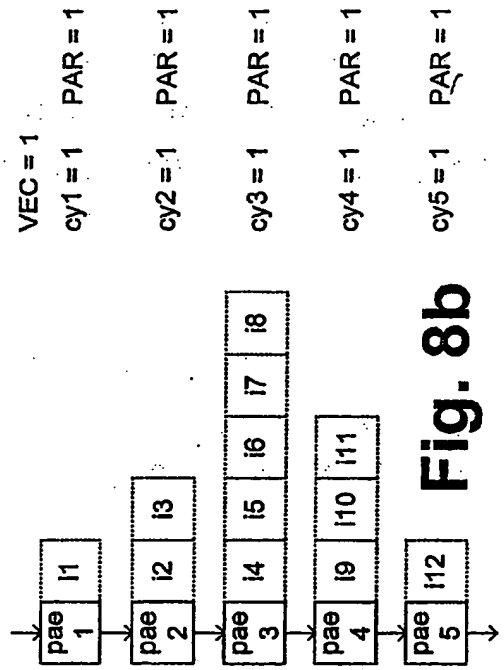


Fig. 8b

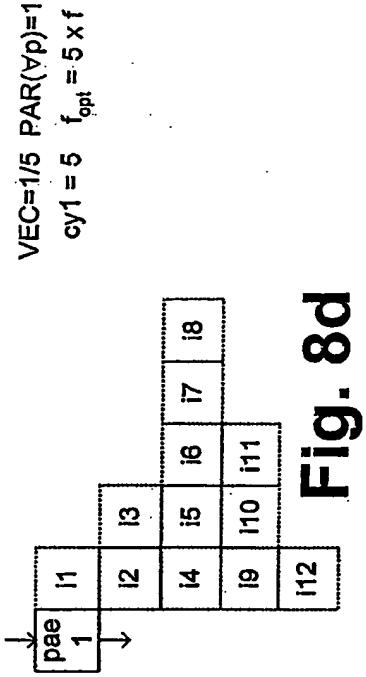


Fig. 8d

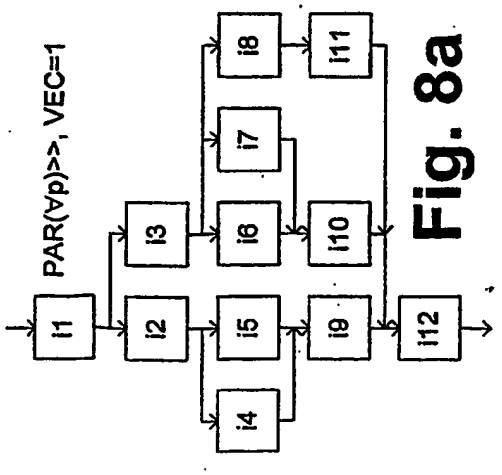


Fig. 8a

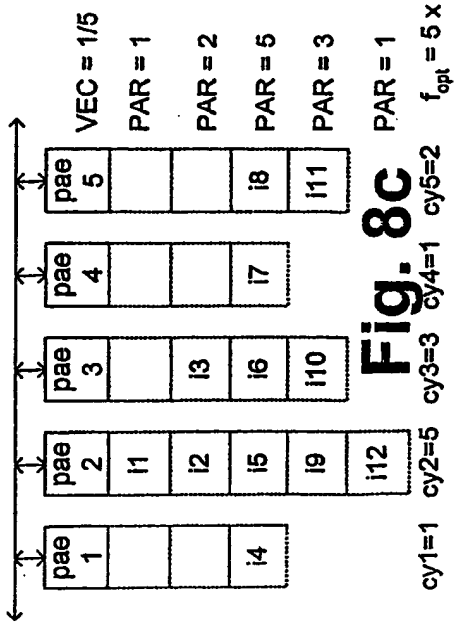


Fig. 8c

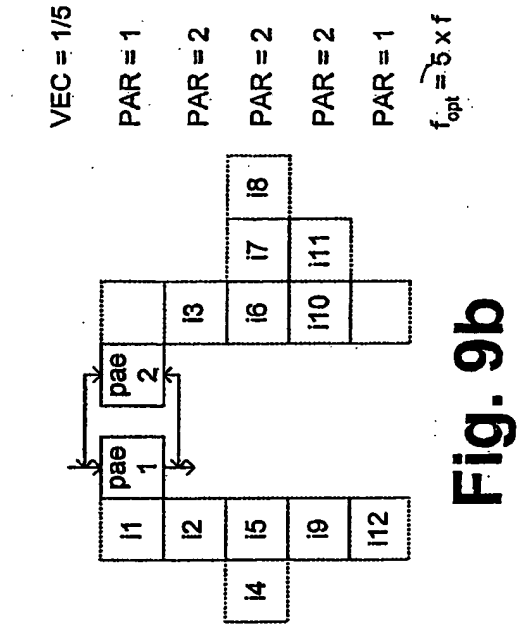

VEC = 1/5
PAR = 1
PAR = 2
PAR = 2
PAR = 2
PAR = 1
$f_{opt} = 5 \times f$
i1
pae 1
i3
pae 2
i4
i2
i5
i6
i8
i7
i9
i10
i11
i12

Fig. 9b

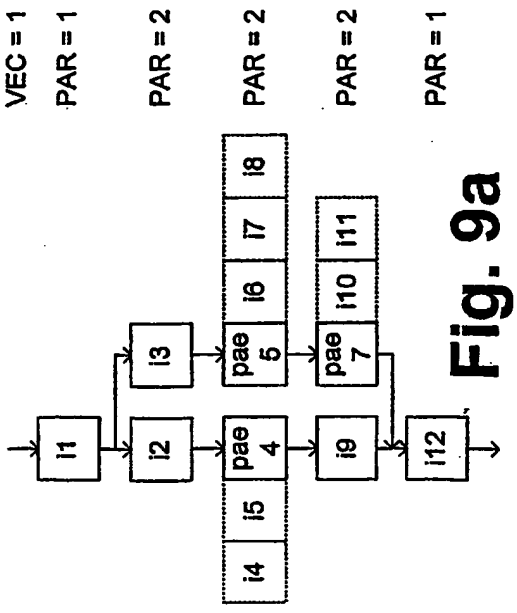

VEC = 1
PAR = 1
PAR = 2
PAR = 2
PAR = 2
PAR = 1
i1
i2
i3
i4
i5
pae 4
pae 5
i6
i8
i7
pae 7
i9
i10
i11
i12

Fig. 9a